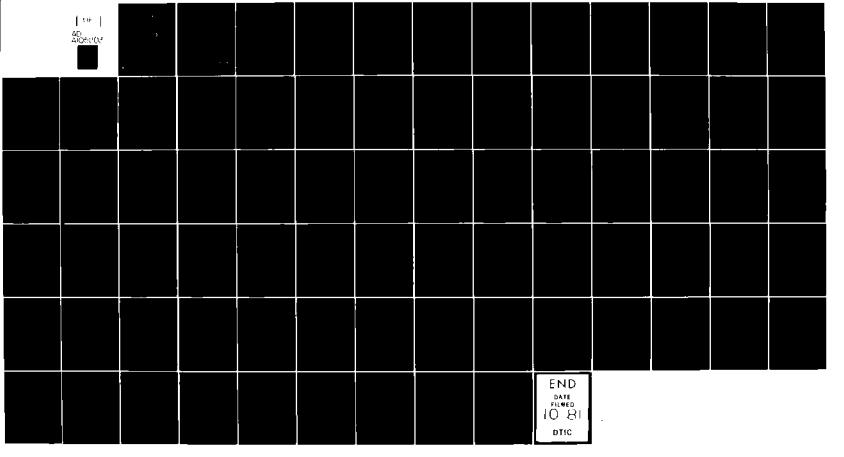
AD-A105 203 KANSAS STATE UNIV MANHATTAN DEPT OF COMPUTER SCIENCE F/G 9/2
COMMAND PROCESSORS FOR DYNAMIC CONTROL OF SOFTWARE CONFIGURATIO--ETC(U)
JUL 80 R FUNDIS, V WALLENTINE DAAG29-78-G-0200
UNCLASSIFIED TR-80-02 NL



END
DATE FILMED
10 81
DTIC

LEVEL II

COMMAND PROCESSORS FOR DYNAMIC CONTROL OF SOFTWARE CONFIGURATIONS

Roxanna Fundis

Virgil Wallentine

Technical Report

TR-80-02

Department of Computer Science
Kansas State University
Manhattan, Kansas

July 1980

This research was supported in part by the Army Institute for Research in Management, Information, and Computer Systems under grant number DAAG@29-78-G-0200 from the Army Research Office.

ABSTRACT

Command language facilities for the construction and execution of software configurations--networks of communicating processes--are very limited today because current operating systems do not support this level of complexity. The Network Adaptable Executive (NADEX) is an operating system which was designed to support dynamic configurations--those configurations which are constructed at command interpretation time--of cooperating processes. These dynamic configurations include arbitrary graphs which may contain cycles. Three command processors have been developed to demonstrate the sufficiency of the NADEX facilities to support dynamic configurations.

NADEX facilities, an overview of the Job Control System, and the command processor configuration environment are presented, followed by user's guides for the command processors. Each command processor has different responsibilities and capabilities for handling configurations: The NADEX Static command processor executes completely connected configurations. The UNIX command processor allows linear configurations to be constructed dynamically, and the MIRACLE command processor allows the dynamic construction of arbitrary configurations. Syntax graphs and sample user sessions are presented for each command processor.

TABLE OF CONTENTS

ILLUSTRATIONS

INTRODUCTION

Command language facilities for the construction and execution of software configurations--networks of communicating processes--are very limited today because current operating systems do not support this level of complexity. The Network Adaptable Executive (NADEX)[11] is an operating system which was designed to support dynamic configurations--those configurations which are constructed at command interpretation time--of cooperating processes. These dynamic configurations may be composed of arbitrary graphs which can contain cycles. Three command processors have been developed to explore the sufficiency of the NADEX facilities to support dynamic configurations. Users' guides to these command processors will be presented in this document, together with syntax graphs and sample user sessions.

A command allows the user to query the state of a program and/or the computer and to manipulate its resources. For example, file maintenance may be achieved through a command such as

```
DELETE MYFILE
```

which deletes a file named MYFILE from the file system; or programs may be executed with a command like

```
PAS32 MYFILE
```

which compiles the Pascal program named MYFILE. The program which reads and interprets these commands as requests to execute operations or other programs is commonly known as a command processor, command interpreter, or shell.

In chapter I, NADEX facilities, an overview of the Job Control System, and the command processor configuration are presented. Much of the basic structure of the NADEX command processors is taken from a command processor called DO which was written by Per Brinch Hansen[3]. This command processor originally ran under the Solo[3] operating system and the parts of it that prove relevant to further command processor development are described in chapter II. Although the Solo operating system does run under NADEX, it is not used for practical purposes. Chapter III contains a description of the NADEX version of DO.

Since the UNIX* shell[2] seems to be the only commercially available command processor that allows the user to dynamically configure commands, a small subset of the UNIX command processor was implemented to test the sufficiency of and demonstrate the use of NADEX facilities. This subset of UNIX is documented in chapter IV. In chapter V, the NADEX implementation of Gray's MIRACLE[5] (Machine Independent Resource Allocation and Control Language) is described. This network command language supports named ports which allows arbitrary configurations to be constructed. Implementation features and conclusions are found in chapter VI. Syntax graphs for the command languages, the NADEX Native Prefix, and sample user sessions are listed in the appendices.

*UNIX is a trademark of Bell Laboratories

CHAPTER I

NADEX FACILITIES

NADEX[11] is a message-based, multi-user, network operating system which was developed in Concurrent Pascal. Most current operating systems support a fixed number of processes cooperating on the execution of a single user task. However, NADEX was specifically designed to support dynamic configurations with a variable number of user processes.

Since this seems to be a rather unpublished field of research[7, 8], there are few guidelines concerning the usefulness of dynamic configurations or how best to utilize them. Several different types of command processors were developed to run under NADEX to research the practicality of dynamic user configurations and the sufficiency of the NADEX Core OS to support them.

The NADEX Core OS[11] provides facilities for realizing configurations. At this level, a configuration consists of processes and explicit communication between these processes. Each process may execute a user program or a system routine. A Data Transmission Stream (DTS) allows processes to communicate.

All NADEX command processors use the NADEX Native Prefix as an interface to the Core OS for access to these facilities. The prefix routines that the command processors use are READ_CHAR, WRITE_CHAR, READ_DATA, WRITE_DATA, READ_PARM, and WRITE_PARM which read and write characters, data blocks (512 bytes), and parameter blocks (32 bytes)

respectively. When a complete Configuration Descriptor (CD) [10] has been created, it is submitted to the Core OS for execution by calling the SUBMIT_CONFIG prefix entry. The NADEX Native Prefix is listed in Appendix B.

Introduction to the Job Control Environment

The Job Control System provides the user interface to the NADEX operating system. It allows the user to operate in a relatively simple environment and to construct configurations in a modular manner. A brief summary of the Job Control System is presented here; more detail may be found in [10].

A user may view a software configuration as being made up of components. Each component may consist of combinations of other components, or subcomponents. A node, which is implemented by a process in the Core OS, is the most primitive component and may perform one of two functions. It may provide access to system resources such as files, devices, or subsystems; or it may execute a user program written in Sequential or Concurrent Pascal. Bi-directional communication between components is achieved by naming the ports [1] of each node that are to be connected and then "READ"ing and "WRITE"ing on these buffered ports. This connection is realized by a DTS.

Each component of a configuration may be described by a Pascal record which is termed a NADEX Partial Configuration Descriptor or PCD[9, 10]. A PCD thus abstracts and encapsulates one or more nodes of a configuration. Each configuration may be composed of one or more PCDs. The information that defines the nodes, ports, and parameters of a particular PCD record is contained in a PCD file. Graphical examples of PCDs are shown in Figures 1 through 6.

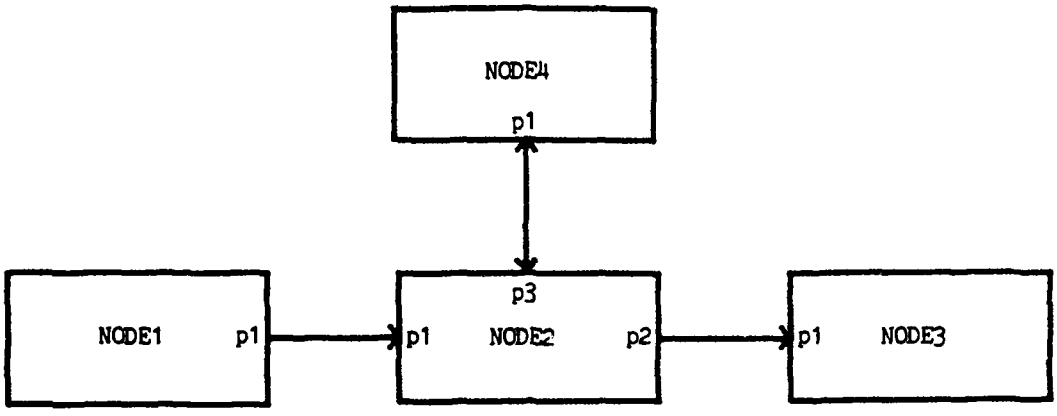


Figure 1. PCD1

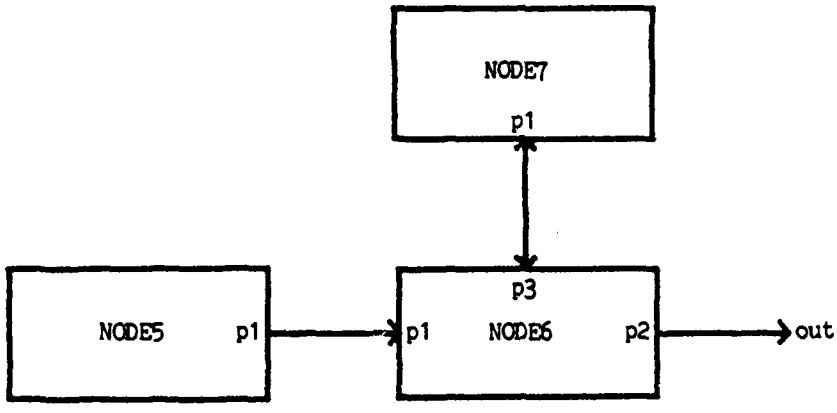


Figure 2. PCD2

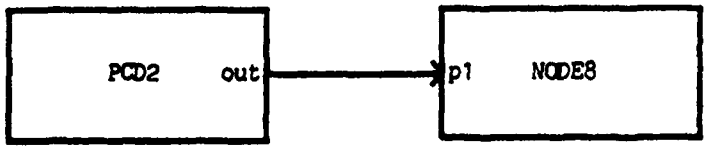


Figure 3. PCDHIER

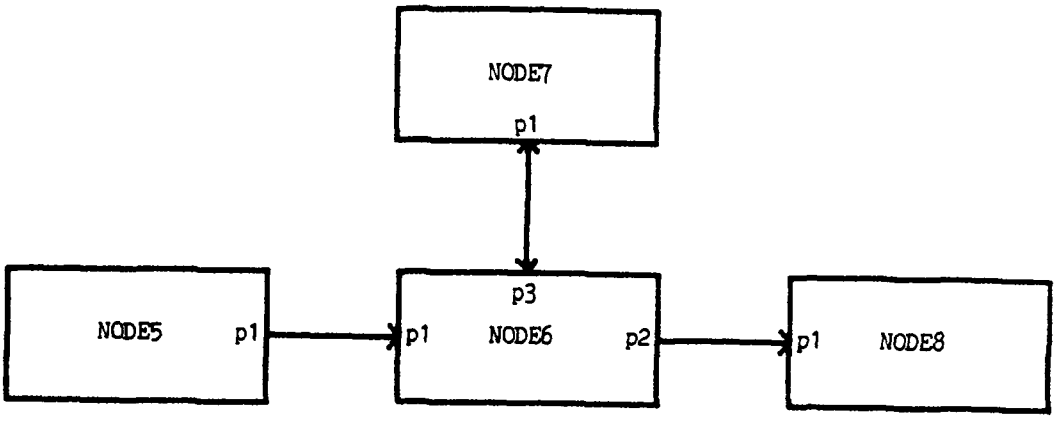


Figure 4. PCDSAME

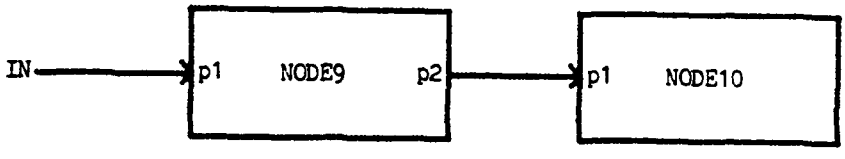


Figure 5. PCD3

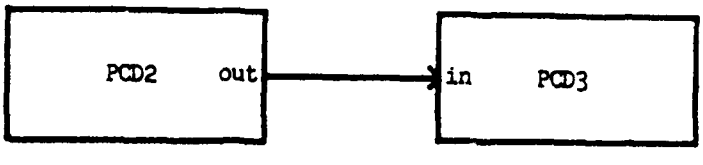


Figure 6. Resulting configuration of PCD2 ! PCD3

PCD1 in Figure 1 is a configuration that contains four nodes. All ports (NODE1.P1, NODE2.P1, etc.) are connected within the PCD so they are internal ports. PCD2 (Figure 2) is a partial configuration which contains three nodes. All are connected internally except NODE6.P2, which is an external port. Its external name is OUT and may be connected to another port either at command time or within a hierarchical PCD as shown in Figure 3.

PCDHIER (Figure 3) is a hierarchical PCD because it contains a component, PCD2, which is itself a PCD. At this level, NODE5, NODE6, and NODE7 are not visible, although PCDHIER will function exactly as PCDSAME in Figure 4 in which only primitive nodes appear.

PCDs may be nested several levels deep; PCDHIER could also become a component of another PCD. The ability to create hierarchial PCDs using the names of "lower level" PCDs eliminates unnecessary duplication where a subcomponent is used more than once. It allows the user to hide certain details (abstraction) and aids in functional decomposition.

When dynamic configuring of nodes is allowed, external ports may be connected at command time. If, for example, a PCD existed such as PCD3 in Figure 5, PCD2 and PCD3 could be connected at command time with a UNIX pipe operator. The command

PCD2 ! PCD3

connects the two external ports and creates the configuration shown in Figure 6. The output from PCD2 becomes the input to PCD3, which in effect connects NODE6.P2 to NODE9.P1.

Besides information about nodes and their port connections, PCDs may also contain templates for parameters which are supplied by the user at command time. A PCD may have parameters of type integer, identifier,

string, boolean, or file descriptor. In addition, parameters may be designated as being mandatory or optional. A default value must be specified for optional parameters.

Only user commands which describe completely resolved configurations--those containing no unresolved ports or parameters--may be executed. PCD1, PCDHIER, and PCDSAME each represent a complete configuration because all ports are connected or resolved and there are no parameters. PCD2 has an external port which is unresolved and therefore represents only a partial configuration.

In order to build a complete configuration, the user must be aware of external ports and parameters. If the PCD has external ports or mandatory parameters that are omitted by the user, a prompt message will be displayed listing the missing information. For example, if the user types the command GREP--a UNIX program that lists lines from an input file that contain a specified string--the following message will be displayed on the user's console:

```
MANDATORY PARAMETER OMITTED

      TRY AGAIN:
        GREP     SOUGHT: STRING
           EXTERNAL PORTS: IN, OUT
```

This message tells the user that GREP expects a string parameter named SOUGHT and that there are two external ports.

Similarly, if external ports are not connected properly, the message

```
ERROR IN PORT CONNECTIONS
```

is displayed, followed by the external port names. Port modes (READ, WRITE, READ-WRITE) and protocols (ASCII, SEQUENTIAL, or RANDOM) must be

compatible. If for instance, a user tries to connect a READ port to another READ port, the command processor will signal an error, but does not currently identify the ports in error. The port mode READ-WRITE is currently implemented as being compatible with either READ, WRITE, or READ-WRITE. ASCII and SEQUENTIAL protocols are compatible with each other. The RANDOM protocol is compatible only with itself. At the present time, the main problem with port connections seems to be one of omission. If the user had problems with port mode or protocol compatibilities, the error message could easily be altered to include the mode and/or protocol as well as each external port name (similar to the type information that is included with parameters).

Command Processor Configuration and Execution Environment

Figure 7 presents an abstract user view of the command processor configuration which runs under NADEX. Each command processor has different responsibilities and capabilities for handling configurations. The NADEX Static command processor will only accept completely connected PCDs. Dynamic connection of ports at command time is not permitted. Its syntax includes both keyword and positional parameters. With the UNIX command processor, unresolved PCDs may be connected to form linear configurations. The MIRACLE command processor allows unresolved PCDs to be connected to create arbitrary graphs. The UNIX and MIRACLE languages that are currently implemented are not necessarily proper subsets of the original command languages; references within this paper to UNIX and MIRACLE refer to the implementions under NADEX.

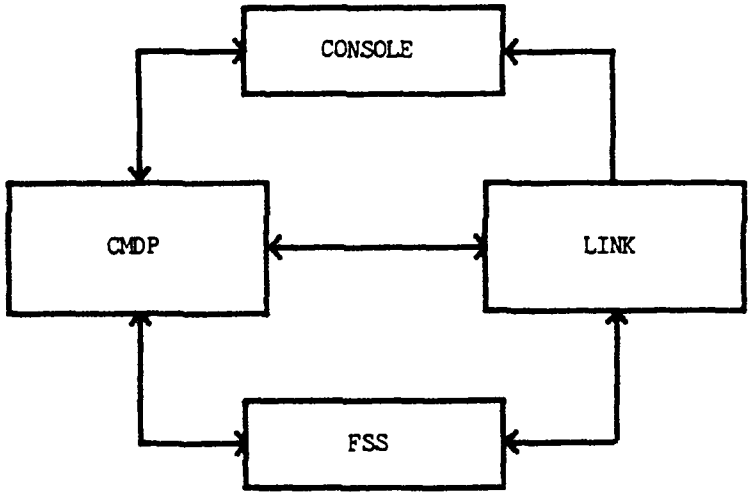


Figure 7. Command processor configuration.

Although each command processor handles a different set of commands, the execution environments are all similar. In general, the command processor parses a command read from the user's console. The existence of PCD and input files is verified, parameters are resolved, output files are created if necessary, and if supported, unresolved ports are connected. This information is stored in "resolved" PCD files. A new PCD is created by the command processor containing the names and connections between the resolved PCDs in the command. Its name is sent as a parameter to the sequential Pascal program LINK[10].

LINK is responsible for looking up all the lower level PCDs, assigning DTSs, assuring resource availability, eliminating user information that is not needed, and doing whatever else is necessary to map the PCD into a CD[10]. LINK then submits the CD to the NADEX Core OS for execution in one of two ways. The first involves a call which causes the command processor configuration to be replaced by the new user configuration. When the user configuration has terminated, the command processor configuration is restored to accept further commands. The second way involves a spin off, where the new configuration executes concurrently with the command processor configuration (providing that resources are available).

Commands that involve text substitution or operations such as allocating, deleting, or listing a file do not require communication with the LINK program and the submission of a new configuration to the operating system; they are handled by the command processor itself by sending messages to the file subsystem.

CHAPTER II

THE DO COMMAND PROCESSOR

The Solo operating system was written by Per Brinch Hansen and only runs a configuration which has an input process, job process, and output process as in Figure 8. The input and output processes may access devices or files by running sequential programs. The job process may only load and execute sequential programs.

The job process initially calls the sequential program DO, which is the command processor. DO reads the user command from the console and parses it. The first part of the command must be an identifier which references an executable program name. A lookup operation is performed on this identifier. If found, and the command contains parameters enclosed in parentheses, the parameters are placed in an argument list. As long as the parameters are either identifiers, integers, or boolean values, no type checking of parameters is done at this time. The number and type of parameters must be checked by each program as it is invoked.

If no errors are detected, the RUN prefix entry is called to load the program referenced within the command. The entire arglist is passed as a parameter. For example, if the command EDIT(CARDS, TAPE) were typed in and if an EDIT program existed, it would be called with a parameter list containing the identifiers 'cards' and 'tape'. It would then be executed within the job process.

Since every program is responsible for checking its own

Figure 8. SOLO configuration.

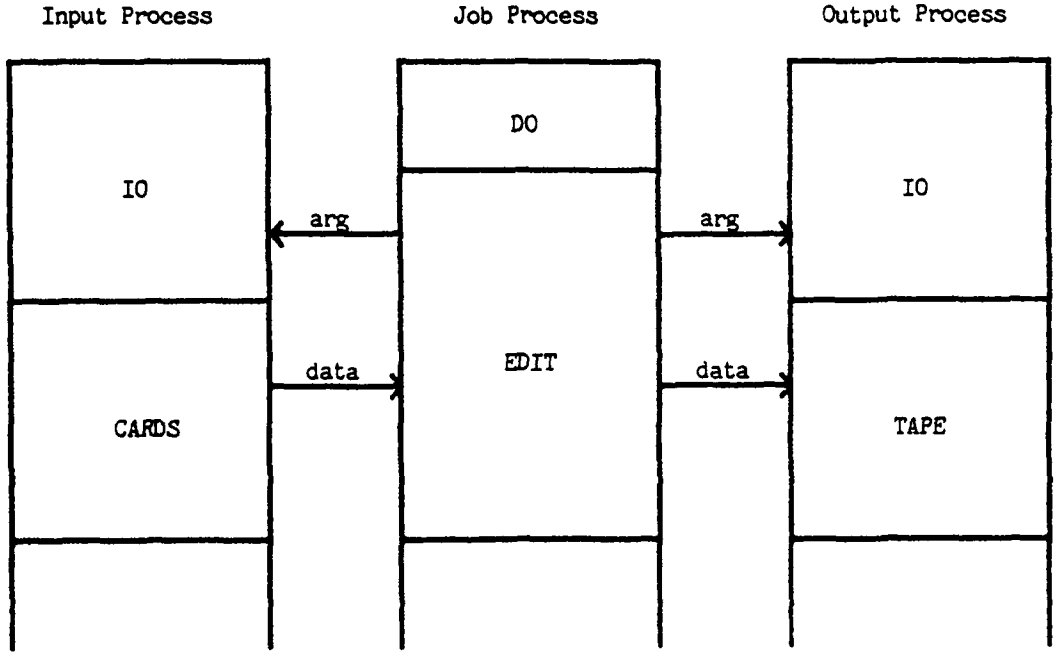


Figure 9. Execution of EDIT(CARDS,TAPE).

parameters, the EDIT program must parse the two parameters and insure that both source and destination filename parameters are names of existing files. If the output file is not found, a new one is created. Since it is the user program that is responsible for configuring nodes, it must tell the input and output processes which programs to run. The EDIT program therefore sends a message containing the first parameter to the IO program executing in the input process. This tells the input process that it is to run the CARDS program. The second parameter is sent to the output process. The output process therefore runs the TAPE program. A diagram illustrating this scenario is shown by Figure 9.

Under SOLO, all configurations are static. The basic shape of the configuration as illustrated by Figure 8 may never vary. The programs that run in those three processes are determined by the name of the user program and its parameters specified in the command.

CHAPTER III

THE NADEX STATIC COMMAND PROCESSOR

The NADEX Static command processor is similar to DO as it only executes static commands. However, while a DO command consists of a sequential program name with its arguments, a similar NADEX Static command processor command consists of the name of a PCD file followed by its arguments. Both the program name listed in the DO command and the PCD file contain information that describe the configuration to be executed. Like DO, the Static command processor does not allow dynamic connections with other nodes at command time. Although the PCDs that are accepted by the Static command processor may contain multiple nodes connected in an arbitrary graph, the connections must be fully resolved within the PCD. All connections must be defined prior to command time.

Unlike DO, this command processor will accept two types of commands. The first type allows "fast" commands to be executed directly while the second type causes configurations to be built and activated. The typical command is of the form

command(arg1,arg2,...,argN)

where "command" is the name of a PCD file describing the configuration to be run or the name of a fast command. The arguments "arg1" through "argn" may be integers, strings, identifiers, booleans, or file descriptors with the following restrictions: Integers must be equal to or smaller than 99999999. Strings must be enclosed in single apostrophes with a length less than or equal to 32 characters. A single

apostrophe may be used within a string by repeating the symbol. Identifiers must be no longer than eight characters. Only the boolean values TRUE and FALSE may be used. Figure 10 illustrates examples of OS/32 MT file descriptors.

```
(a)  SYS2:A.PAS/G
(b)  B/P
(c)  C.OBJ
(d)  D.PAS/S
(e)  E
(f)  USR6:F.CSS
```

Figure 10. Examples of OS/32 MT file descriptors.

Figure 10a is interpreted as: Volume SYS2:, file name A, extension .PAS, on the user's group account. 10b names the file B (with a blank extension) on the user's private account. The volume is assumed to be the user's current volume. Figures 10c through 10f may be interpreted similarly. An account of S indicates the system account; if this field is omitted, the default value is P for private.

Fast Commands

The six fast commands (HELP, ALLOCATE, DELETE, RENAME, ATTR, LIST, and SIGNOFF) do not start up new configurations. The function and expected arguments of each of the fast commands will be discussed and examples provided.

The HELP command accepts no arguments. It displays a list of the fast commands, their arguments, and functions at the user's console. Also displayed is a list of executable configuration names and their functions.

ALLOCATE is used to create a new file. It requires one parameter, the name of a new OS/32 MT file descriptor. If the specified file

already exists, an error message will be displayed.

DELETE also requires a parameter consisting of OS/32 MT file descriptor. If the designated file does not exist, an error message is displayed.

The RENAME command requires two file names separated by commas: the first one must be the name of an existing file and the second must be a the name of a new file. If the first name does not exist or the second one does exist, an error will occur and no names will be changed.

The fast command ATTR requires a file descriptor parameter. It returns the attributes of the specified file, including the fully qualified name, the number of records, and length.

The entire contents of a file may be displayed at the console by typing LIST and a file descriptor. If the file does not exist, an error will occur.

To terminate a user session, the fast command SIGNOFF is used.

Examples of fast commands are provided in Figure 11.

```
HELP
ALLOCATE NEW.PAS
ALLOCATE SYS2:TEST.PAS/P
DELETE NEW.OBJ
DELETE USR6:FILE3.TXT
RENAME FIRST.PAS,SECOND.PAS
RENAME SYS2:WOW.OBJ,WOWOWOW.OBJ
ATTR SYS2:WHAT.PAS
ATTR HOW.OBJ/P
LIST TELL.TXT
LIST LISTS.TXT
SIGNOFF
```

Figure 11. NADEX Static fast commands.

Configuration-Building Commands

If the command is not a fast command, the first item in the command line must be the name of a PCD file. This PCD file is read by the command processor to determine what mandatory and optional arguments are expected. The arguments typed in are parsed, and if they are compatible with the templates, they are filled into the PCD and sent to LINK for execution as explained in chapter I.

Only one command per line is accepted. An optional semicolon may follow the command. Blanks may be used anywhere except within an identifier or a number. Parentheses surrounding the argument list are optional. Arguments must be separated by commas. A NEWLINE character may directly follow a comma in an argument list. Association between actual parameters (those typed in by the user at command time) and formal parameters (the templates specified in the PCD) may be made either positionally or by explicit naming. To illustrate the discussion of parameters, a PCD named PARMEX has been defined as:

```
PARMEX  [PARM1 : STRING DEFAULT('NS')],
        [PARM2 : FD DEFAULT(SYS2:IN.TXT/P)],
         PARM3 : FD,
        [PARM4 : INTEGER DEFAULT(48)]
```

PARMEX may use 4 parameters, 3 of which are optional. The first parameter which is optional and named PARM1, is a string type and has a default value of 'NS' if it is not specified at command time. The next two parameters are file descriptors. PARM2 is an optional parameter with a default value of SYS2:IN.TXT/P. PARM3 is a mandatory parameter; failure to specify a value at command time constitutes an error. The last parameter, PARM4, is an optional integer with a default value of 48. Figure 12 lists possible commands involving PARMEX.

If specified positionally, the order of the actual parameters must

correspond to that of the formal parameters. Optional arguments that are omitted must be indicated by commas. Figure 12a indicates that the first and second parameters are omitted and that the value of the third parameter is OUT.TXT. Optional parameters omitted at the end of a parameter list do not need to be indicated with commas, so the last comma is acceptable but not necessary. Figure 12b will therefore have exactly the same effect as the previous command. When named parameters are used, commas do not need to be used to indicate positions of parameters which have been omitted. Figure 12c is also equivalent to the previous two commands.

Named keywords may be specified in any order as illustrated by Figure 12d. The same values listed positionally are given by 12e. A combination of positional and named parameters may be used, but it should be noted that the use of a named parameter will reset the positional pointer. Notice that in Figure 12f, the integer 55 is in the third position but it becomes the value for the fourth parameter because it immediately follows a named occurrence of the third parameter. In Figure 12g, the attempt to place the value EXTRA in a fifth parameter will result in an error because PARMEX only has four parameters.

Parameter values specified at command time may be overridden with the use of keywords as in Figure 12f where PARM1 first gets a value of 'DOG', and later receives the value 'COW' with the use of the keyword parameter. Although this is legal, it is not good programming and a warning message will be issued by the command processor.

```
(a)     PARMEX (,,OUT.TXT,);
(b)     PARMEX ,,OUT.TXT
(c)     PARMEX (PARM3 = OUT.TXT);

(d)     PARMEX (PARM4=312,PARM2=RED.TXT,PARM1='SS',PARM3=TEST.PAS);
(e)     PARMEX ('SS', RED.TXT, TEST.PAS, 312)

(f)     PARMEX ('DOG',PARM3=OUT.TXT,55,PARM1='COW')

(g)     PARMEX PARM4=999, EXTRA
```

Figure 12. NADEX Static commands with parameters.

CHAPTER IV

THE UNIX COMMAND PROCESSOR

The UNIX Command Language

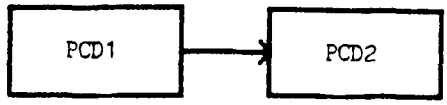
The UNIX command processor developed to run under the NADEX OS implements linear connections between PCDs using a UNIX style syntax. As with the Static command processor, it will accept either "fast" commands or configuration-building commands. A basic UNIX configuration-building command consists of an identifier which is a PCD name followed by its parameters:

PCDNAME parm1 ... parmN

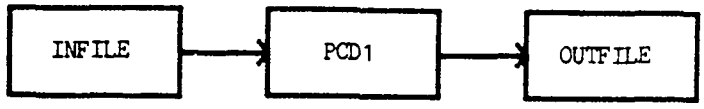
No parentheses surround the parameter list. Parameters must be separated with blanks. Keyword parameters are not accepted.

UNIX Configuration-Building Operators

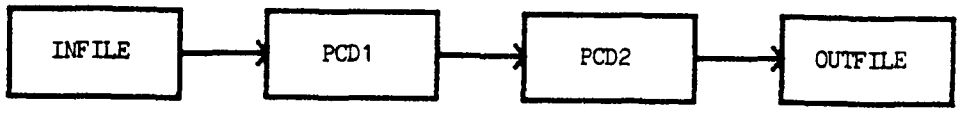
There are four operators that may be used to connect the unresolved ports of PCDs. Figure 13 illustrates the use of these operators in UNIX commands with the resulting configurations. The first operator is the UNIX pipe (!) which connects the external ports of two unresolved PCDs as in Figure 13a. Each PCD can have a maximum of two external ports--one for input from and one for output to a pipe. This is necessary since the standard UNIX shell does not support specification of port connections in the command language. Input and output redirection is supported with the < and > operators respectively (Figure 13b - c). Output from a PCD may be appended to a file by using



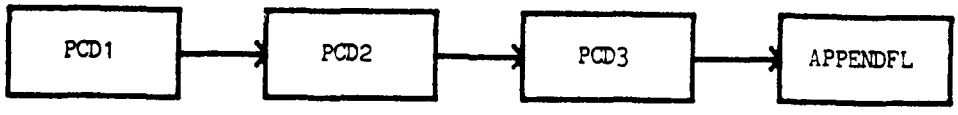
(a) PCD1 ! PCD2



(b) PCD1 <INFILE >OUTFILE



(c) PCD1 <INFILE ! PCD2 >OUTFILE



(d) PCD1 ! PCD2 ! PCD3 >>APPENDFL

Figure 13. UNIX commands and resulting configurations.

the >> operator as in Figure 13d. Each input, output, and append operator must be directly followed by a file name (see later discussion on the hierarchical file system).

In Figure 13a, the output port from PCD1 is connected to the input port of PCD2 with the pipe operator. In Figure 13b, PCD1 has two external ports. Its input port will be connected to a file named INFILE and the output will be channeled to a file named OUTFILE. The INFILE and OUTFILE nodes in the corresponding configuration are fileaccess nodes, which are nodes that provide access to named files.

With a maximum of one input and one output port, only linear configurations (pipelines) can be built. The following commands violate this rule and therefore are illegal:

```
PCD1 <INFILE1  <INFILE2            (two input files)
PCD1 >OUTFILE1  >>APPENDFILE       (two output files)
PCD1 <INFILE  >OUTFILE ! PCD2      (two output files)
PCD1 ! PCD2 <INFILE                (two input files)
```

Command Sequencing Operators

The &, ;, &&, and !! operators also have special meaning to the UNIX command processor. Examples are provided in Figure 14 below. When an ampersand (&) follows a command, the command will be spun off. The command processor configuration will not terminate; new commands may be entered and executed concurrently with this configuration.

The semicolon (;) may be used to terminate a command. It is not required when a single command is entered on a line as the NEWLINE character will signal the end of the command. However, when more than one command is entered on a line as in Figure 14b, a semicolon must separate the two commands. If a line contains multiple commands and an error is detected (the syntax is wrong, a file is not found, mandatory

parameters are omitted, or there is an error in the external port connections) the rest of the line is ignored. For example, if

```
WC <INFILE  >OUTFILE; LIST OUTFILE
```

was typed and INFILE does not exist, the second command on the line, LIST OUTFILE, will be ignored.

The McCarthy "andf" (&&) and "orf" (!!) operators provide conditional execution of commands[2]. When an "andf" is used, the command on the right of the && will be executed only if the first command was successful and is similar to

```
IF cmd1 THEN cmd2 .
```

The "orf" operator allows execution of the second command to proceed only if the first was unsuccessful as

```
IF NOT cmd1 THEN cmd2 .
```

```
(a) PCD4 ! PCD5 >OUTFILE &
(b) PCD1 >OUTFILE ; PCD4 ! PCD5
(c) COPY <OLDFILE  >NEWFILE && DELETE OLDFILE
(d) PCDNAME <THISFILE  !!  PCDNAME  >THATFILE
(e) PCD1 !! PCD2 && PCD3
```

Figure 14. UNIX commands using sequencing operators.

UNIX Fast Commands

The UNIX command processor accepts twelve fast commands which are CREATE, CREATED, DELETE, ADD_FD, RENAME, ATTR, LIST, HELP, LS, CWD, PWD, and SIGNOFF. Many of these fast commands are similar to those of the Static command processor, except that UNIX path names are used instead of OS/32 MT file descriptors. The syntax graph for a path name is found in the UNIX syntax graphs (Appendix A); a discussion of the semantics of path names is found in [2, 4]. Path names are illustrated below.

```
/DIR1/DIR2/DIR3/TARGET
^TEST
^^^MYFILE
YOURFILE
DIRV/FILEX
```

A slash (/) that begins a path name indicates the root directory. Slashes are also used to separate intermediate directories. The last identifier in a path name is the target name which may be either a directory or a data file name. A carat (^) indicates the parent directory.

The CREATE command allocates a new ASCII file and places its name in a UNIX directory. It requires one argument, the path name of the file to be allocated. If the file already exists or an intermediate directory is missing, an error will result.

CREATED allocates a new directory file. It requires the path name of a new directory file. An error occurs if the designated file already exists or an intermediate directory is missing.

DELETE is used to delete the specified file and erase its name from the UNIX directory. It requires one argument consisting of the path name of an existing file. If the designated file does not exist, an error message is displayed.

ADD_FD is used to make an existing OS/32 MT file accessible to the UNIX directory. It requires two arguments: the first must be the new path name and the second must be the name of the existing MT file enclosed in apostrophes. If the path name already exists or the MT file does not exist, an error occurs.

RENAME, ATTR, LIST, HELP, and SIGNOFF work exactly the same as they do in the Static command processor, except that they require path names as arguments instead of OS/32 MT file descriptors.

LS lists the names of all the files within the current working directory at the console. It accepts no arguments.

CWD is used to change the current working directory to that of the path name specified as the argument.

PWD displays (prints) at the console the fully qualified path name of the current working directory.

Figure 15 lists examples of UNIX fast commands.

```
CREATE PCDS/GREP
CREATE GREP/PAS
CREATE OBJ
CREATED /PAS
CREATED PAS/HI
DELETE /PCDS/GREP
DELETE ^^^WC
ADD_FD /PCDS/GREP 'GREP.PCD'
ADD_FD WC 'WC.PAS'
RENAME PCDS/UNIX PCDS/OLDUNIX
ATTR UNIX
ATTR ^TEST
LIST /PCDS/WC
LIST GREP
HELP
LS
CWD /PCDS
CWD /
CWD ^^
PWD
SIGNOFF
```

Figure 15. Examples of UNIX fast commands.

Dynamic Construction of Linear Configurations

As the name of a PCD file is encountered in a command line, that file is read in and its templates for parameters and external connections are matched with the values supplied in the command. If no errors are detected, the missing parameters (if any) are filled in and the "resolved" PCD file is then written out with an extension of .RES. A unique number is appended to its file name so that no confusion will

arise if the command contains multiple instances of a PCD, each with different parameters (see following example). At this time, the command processor creates a hierarchical node in a new PCD which "remembers" the name of the file just written out. This new command PCD will represent the configuration created at command time. It will contain the names of all the PCDs found in the command line as well as fileaccess nodes for pipe input and pipe output (<,>) filenames.

After a node has been created in the command PCD for each PCD and pipe input or output file in the command, all ports are connected. Ports that were external to the lower level PCDs are now internal to the new PCD created by the command processor and are therefore able to be connected at this time. This fully resolved PCD is given the name CMD.RES and written out. Its name is then sent to the LINK program, which will read in and later delete each RES file, assign DTSs, and complete the configuration before submitting it.

Given the existence of GREP and WC* PCD files, a user could type in the command line

```
GREP </TXT/HELP 'EXT' ! GREP 'PORTS' ! WC  >/TXT/OUTFILE .
```

The command processor would first read in the GREP PCD file. It would assign the string 'EXT' to the parameter named SOUGHT and then look up the input file HELP to make sure it is there. The command processor is responsible for verifying the existence of all input files appearing in the command line since this is a likely source of user error. If the input file exists, a fileaccess node is created to read in the file HELP and this becomes the first node in the command PCD(Figure 16a).

---

*WC is a UNIX word count program that prints the number of words, lines, and characters found in its input file.

Since this file name was preceded by a pipe input (<) character, the port name for this fileaccess node is named 'OUT' so that it can be connected to the port GREP.IN. After the pipe character is detected and all arguments are resolved for the GREP PCD, the file is written out with the unique name GREP1.RES (Figure 17a) and a hierarchical node referencing the GREP1.RES PCD file is allocated in the command PCD as shown in Figure 16b.

The next PCD is then read in, which is GREP in this example. The parameter is now 'PORTS' so this information is filled into the PCD, and the resolved PCD is written out with the name GREP2.RES (Figure 17b) and becomes the third node in the command PCD (Figure 16c).

Similarly, a fileaccess node is created with a port named 'IN' for the output file OUTFILE (Figure 16d) and after the PCD for WC has been resolved, WC3.RES is written and assigned to a node. The new PCD now contains five nodes of Figure 16. After all ports have successfully been connected, the complete command configuration is illustrated by Figure 18.

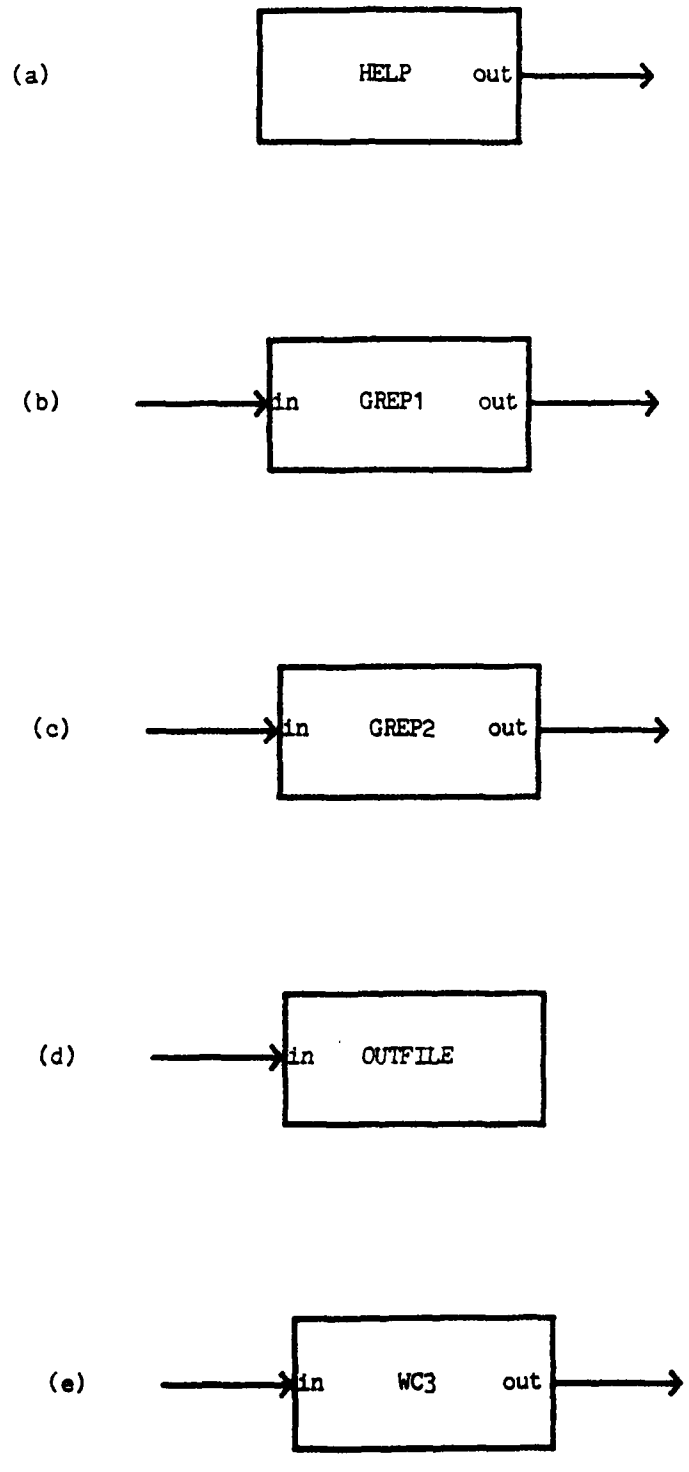
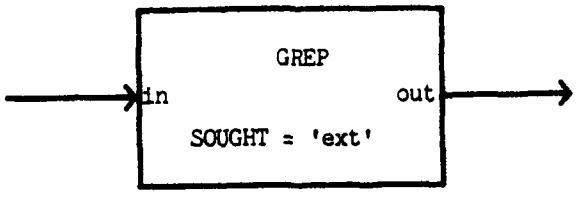
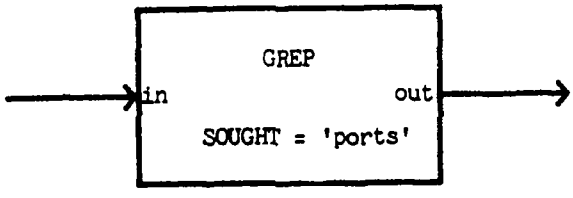


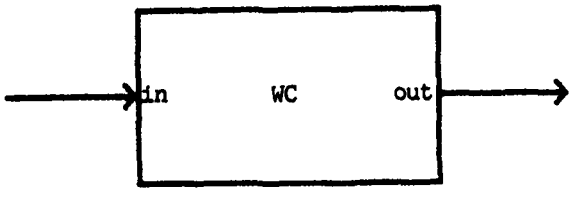
Figure 16. Hierarchical nodes in CMD.RES



(a) GREP1.RES



(b) GREP2.RES



(c) WC3.RES

Figure 17. Resolved PCD files written out by the UNIX command processor.

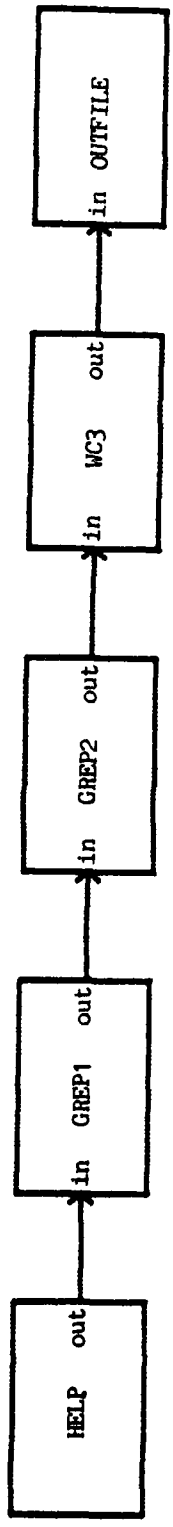


Figure 18. Completed command configuration.

CHAPTER V

THE MIRACLE COMMAND PROCESSOR

MIRACLE[4] (Machine Independent Resource Allocation and Control Language) was chosen for the implementation of dynamic NADEX configurations allowing arbitrary graphs because it supports named ports. It is an expression-oriented language and its command syntax is similar in many respects to UNIX except that the use of named ports within the language no longer places a one-in one-out linear restriction on the connection of PCDs.

MIRACLE commands may either construct configurations or invoke internal functions. Those commands which build configurations reference PCDs which are external to the command processor. Internal functions reference built-in functions ("fast" commands) or strings containing MIRACLE expressions. These internal functions begin with a '$' to distinguish them from external PCDs with the same name.

Configuration-building commands

As with UNIX, the basic form of a MIRACLE configuration-building command consists of a PCD name followed by its parameters:

PCDNAME parm1 ... parmN

Either positional or named parameters may be used.

If the PCD has external ports, all such ports must be connected at command time with one of the port connector operators. The simplex input (<), simplex output (>), append (>>), and implicit pipe (!) are

familiar to UNIX users. In addition, MIRACLE defines several other port-connecting operators, of which the update (<>) and explicit binding (!!) operators are currently implemented. Each <, >, <>, or >> operator must be directly preceded by a port name (all port names within a PCD must be unique) and followed by a pathname or *<connection number> such as "*1". The *<connection number> construct represents a user-defined link or data path between two ports.

If data from a file serves as input or output for a PCD, the PCD port must be named, followed by the port operator and the path name of the file:

PCDA P1<INFILE P2<>RWFL P3>OUTFILE P4>>APPFILE

In this example, PCDA has four external ports. P1 is an input port that will read data from a file named INFILE. Data may be read and/or written to the file named RWFL through port P2. P3 will serve as an output port for data to be written to the file OUTFILE and data will be appended to the file APPFILE through port P4.

As with UNIX, when a linear configuration is built, the implicit pipeline operator (!) can be used:

PCD1 ! PCD2

The implicit operator can only be used to connect a PCD that has a single output port to another PCD with a single input port. Therefore, port names are not necessary for linear configurations.

However, linear connections can also be described with MIRACLE syntax in a more explicit manner by naming the ports and connection numbers:

PCD1 OUT>*1 !! PCD2 IN<*1

The explicit port connecting operator (!!) tells the command processor

that all connections will be expressly defined. In the above example, the port PCD1.OUT is to be connected to port PCD2.IN over connection number 1. When named ports are used, configurations are no longer restricted to being linear. There is a system-defined limit of 16 connections. (A maximum of 32 ports may be connected at command time.) Exactly two ports may be connected with the same connection number; an attempt to assign more than two or only a single port to a specific connection number will be detected by the command processor as an error.

The explicit connector must always be used when any PCD within a configuration has more than one input or output port. No mixing of implicit and explict connections is allowed. The following command is illegal

```
PCD1 ! PCD2 IN<*3  !!  PCD4 OUT>*3
```

because PCD2 obviously has two input ports-only one of which is explicitly connected.

```
   UNIX:  COPY <FILE1  >FILE2
MIRACLE:  COPY IN<FILE1  OUTPUT>FILE2

   UNIX:  WC <INFILE ! CONSOLE
MIRACLE:  WC IN<INFILE ! CONSOLE
     or   WC IN<INFILE  OUT>*4 !! CONSOLE  IN<*4

   UNIX:  no equivalent
MIRACLE:  PCD1  PORT3<*2  PORT2<*4  PORT1>>*1
```

Figure 19. Comparison between UNIX and MIRACLE commands.

All configurations that can be built with the UNIX command processor will also run under the MIRACLE command processor with the addition of port names where appropriate (see Figure 19). However, the reverse is not true if the configuration is not linear. Figures 20 and

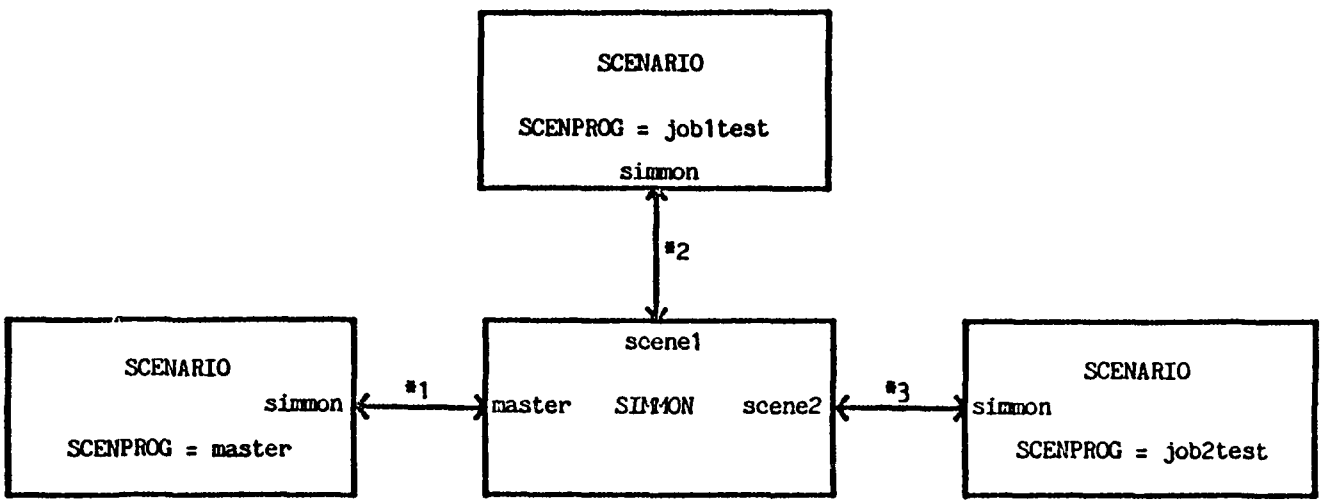
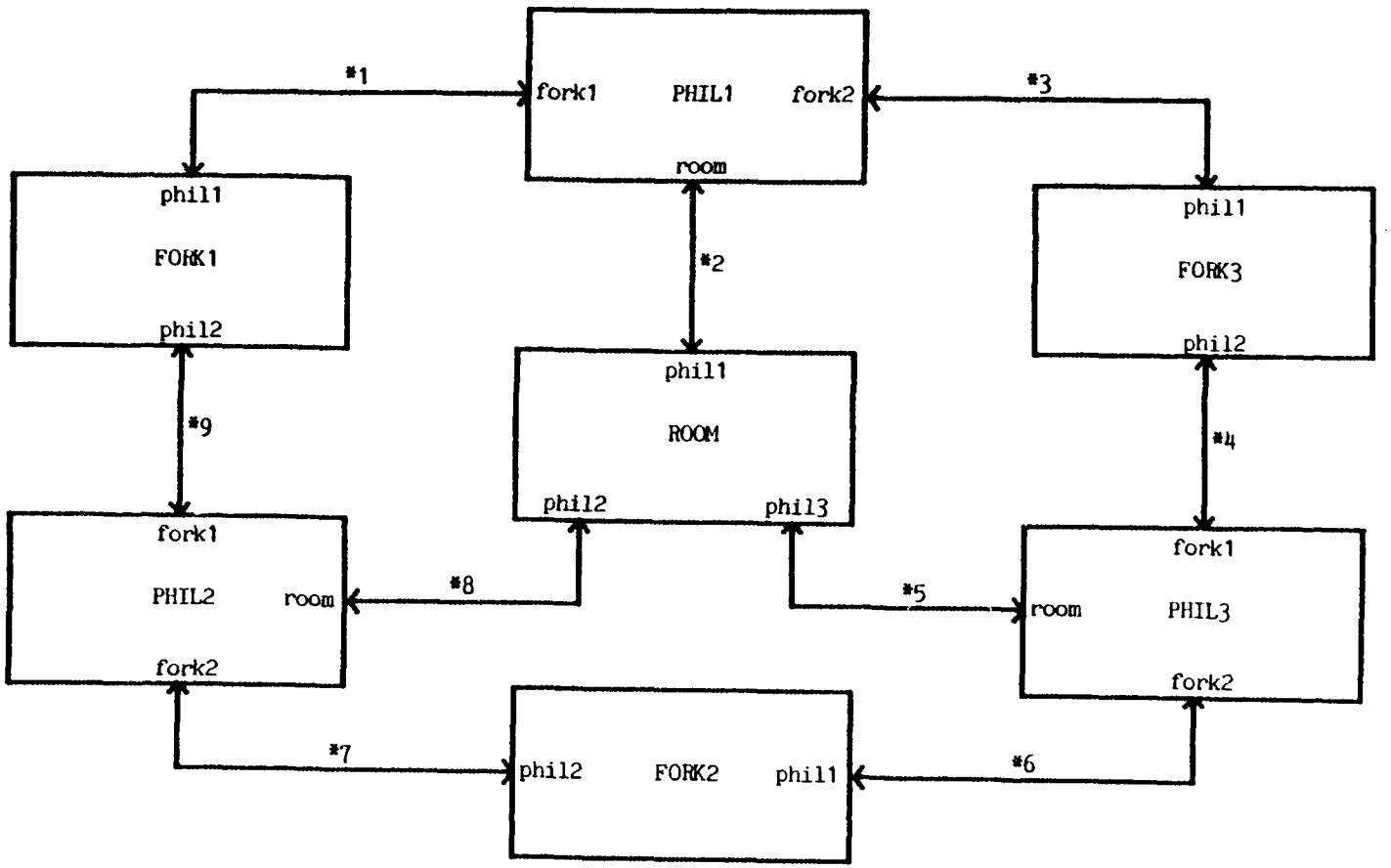


Figure 20. SIMMON MASTER<>#1 SCENE1<>#2 SCENE2<>#3 !! SCENARIO MASTER SIMMON<>#1 !! \
SCENARIO JOB1TEST SIMMON<>#2 !! SCENARIO JOB2TEST SIMMON<>#3




```
PHIL1 FORK1<>*1 ROOM<>*2 FORK2<>*3 !! FORK1 PHIL1<>*1 PHIL2<>*9 !! \
PHIL2 FORK1<>*9 ROOM<>*8 FORK2<>*7 !! FORK2 PHIL1<>*6 PHIL2<>*7 !! \
PHIL3 FORK1<>*4 ROOM<>*5 FORK2<>*6 !! FORK3 PHIL1<>*3 PHIL2<>*4 !! \
ROOM PHIL1<>*2 PHIL2<>*8 PHIL3<>*5
```

Figure 21. Dining Philosophers.

21 demonstrate two MIRACLE commands and the configurations which result. The configuration shown in Figure 20 will run a simulation program. The Dining Philosophers configuration is illustrated in Figure 21. Neither could be built at command time with the UNIX command processor because they are not linear configurations.

Internal Functions

Besides building a configuration which the command processor must send to NADEX for execution, the user may invoke internal functions which are evaluated within the command processor. The names of these internal functions must be preceded by a dollar sign ($) to distinguish them from external names (PCDs). The basic form for a function invocation is:

$funct_name p1 ... pN

where "funct_name" is the name of a built-in (primitive) function or previously stored string.

Built-in functions. Many of the MIRACLE built-in functions are similar to the "fast" commands implemented in the two command processors described previously. Those which are currently implemented in MIRACLE are described below with examples provided in Figure 22.

ALLOC and ALLOCD operate the same as the UNIX CREATE and CREATED, respectively. DELETE, ADD_FD, ATTR, LIST, LS, CWD, PWD, and SIGNOFF operate the same as their UNIX counterparts.

The internal function HELP ($HELP) displays at the console the names of MIRACLE fast commands; the external command HELP (with no $) causes a configuration to be executed which displays executable external names (PCDs).

The internal function DCL allows the user to declare an internal variable of type string or boolean. The syntax for this command is:

$DCL $var_name: var_type

where "var_name" is a unique internal variable and "var_type" is either STRING or BOOLEAN.

EDIT displays the contents of an internal string variable at the console. It requires one argument which is the name of the string variable.

The internal function NOT will negate the result of the parenthesized expression that follows it. If, for example, the expression is

$NOT ($ATTR /MIRC/PCDS)

and the pathname exists, the parenthesized expression result is TRUE. The NOT function will negate this result--the result of the entire expression is FALSE.

```
$ALLOC /THIS/NEWFILE
$ALLOCD ^THAT/NEWDR
$DELETE /THIS/USELESS/FILE
$ADD_FD ADDEE 'ADDOR.PAS'
$ATTR IS/IT/REALLY/THERE
$LIST ^^GROCERIES
$EDIT $SIMSEQ
$HELP
$CWD /TOOLS
$PWD
$LS
$DCL $A : BOOLEAN
$DCL $STRV : STRING
$NOT ($CWD ^OLDDR)
$NOT ($PWD)
$NOT (PCD2)
$NOT (PCD1 ! PCD4)
$SIGNOFF
```

Figure 22. Examples of MIRACLE fast commands.

Internal string functions. A user-defined internal string function may be executed by typing its name followed by its arguments. The command string may be stored with the construct

```
$INTFN := ' e '
```

where "e" is any valid MIRACLE expression, including other internal functions or configuration-building commands.

Either positional or keyword parameters may be included in the command string. A pound sign (#) followed by a number indicates a positional parameter. A pound sign followed by an identifier may be used for named parameters. For example:

```
$FN1 := '$CWD #1; $LS'
```

stores the string containing two built-in functions in the internal function variable FN1. This function may be invoked with the construct

```
$FN1 /MIRC/PCDS
```

which causes the command processor to evaluate the contents of the expression stored in the string. /MIRC/PCDS will be matched with the argument required by $CWD. The current working directory will be changed to /MIRC/PCDS and LS will cause the contents of this directory to be listed at the console.

Similarly, the internal string function FN2 may be initialized and later invoked with the following command sequence:

```
$FN2 := '$CWD #PN;  WC IN<#2  OUT>OUTFILE'
$FN2  PN=/PCDS  GREP/PAS
```

Keyword parameters are counted and may reset the positional counter. In the preceding example, the named parameter PN is counted as parameter number one and the parameter GREP/PAS is parameter number two.

Sequencing operators

The semicolon (;), NEWLINE, and independent fork (&&) characters are binary operators which must be directly preceded and directly followed by an expression. Thus, a command such as

```
$LS;
```

followed by a NEWLINE character will result in a

```
SEQUENCING OPERAND ERROR
```

because the NEWLINE directly followed the semicolon. The command

```
$LS; $PWD
```

is perfectly acceptable; the semicolon is a binary operator which separates the two expressions.

The independent fork (&&) is a spin off. Gray also included dependent forks (&) but these were not implemented.

Commands may occupy multiple lines by directly preceding the NEWLINE character by a backslash (\) to escape the NEWLINE. For example,

```
$CWD
/DR/PCD
```

will result in an error because of the NEWLINE between $CWD and its argument. However, in the following two examples, the NEWLINE has been preceded by the backslash character, causing the NEWLINE to be ignored.

```
$CWD \
/DR/PCD

$LS; \
$PWD
```

The use of the escaped NEWLINE is particularly useful when commands are very long (as in Figures 20-21).

Expression results and control structures

Since MIRACLE is an expression-oriented language, every expression returns a result. However, only expressions on the right hand side of a sequencing operator are remembered; left hand side results are effectively ignored.

MIRACLE supports IF-THEN-ELSE-FI, EXIT, and LOOP control structures. The reserved words used in these control structures must be preceded by a period (.) to distinguish them from external names or internal functions.

IF expressions are of the form:

.IF e .THEN e .FI or .IF e .THEN e .ELSE e .FI

where "e" is any valid MIRACLE expression. (IF expressions can thus be nested.) Since one of the productions for an expression is a binary operator (such as a NEWLINE) separating two expressions, the following construct can be produced:

```
.IF $pwd
$ls .THEN pcd parm1 .FI
```

A more useful command sequence might be:

```
.IF pas32 <testprog >errfile \
  .THEN .IF pestab testprog .THEN testprog .FI \
  .ELSE $list errfile; pedit <testprog .FI
```

The first expression would run the Pascal compiler configuration with input from a program named TESTPROG. Output from the compiler would be channeled to a file named ERRFILE. Assuming that the compiler was modified to produce a return code that indicated when any errors were detected, the result of the IF expression would determine further action. If no errors were found, the value of the first expression would be TRUE so the THEN-clause would be executed. If the PESTAB operation on TESTPROG was successful, the program would then be

executed. However, it an error was detected during the compilation, the ERRFILE (containing the list of detected errors) would be listed at the console and the program opened for editing.

.EXIT must be followed by a boolean TRUE or FALSE and is used to terminate a function. The boolean value then becomes the value of the expression. For example:

```
$fn3 := ' .IF $not $attr myfile \
          .THEN .EXIT FALSE \
          .ELSE edit myfile .FI '
```

The .LOOP control structure is terminated with the reserved word .END . Used within a loop, the reserved word .NEXT passes control back to the beginning of the loop. As with .LEAVE, its appearance is only valid within a loop. The reserved word .LEAVE must be followed by an expression. It exits the loop, and the value of the expression following it becomes the value of the loop expression.

CHAPTER VI

SUMMARY AND CONCLUSIONS

The development of the Static, UNIX, and MIRACLE prototype command processors was motivated by four objectives. The investigation of the sufficiency of the NADEX operating system to support arbitrary dynamic configurations was a primary objective. Under the UNIX command processor, linear dynamic configurations were realized. With the MIRACLE syntax allowing the user to specify ports and connections between these ports, arbitrary configurations were constructed at command interpretation time and successfully executed, thus proving the sufficiency of the NADEX OS.

A second area of investigation was the sufficiency of PCDs. The information needed to implement any of the command processor features was found to be present in the PCD record, as well as information necessary to provide the user with help messages. Not all command processors used every field in a PCD record (named parameters are not used in UNIX, for example) but no information was found to be lacking.

The development of portable command processors was the third objective. These three command processors can be used with any operating system that meets three criteria: The operating system must be able to realize configurations represented by PCDs[9]. The system must be interactive, allowing the user and the command processor to communicate. Finally, the operating system must support a file system.

Modifications to the command processors may be required to interface to the file and console systems, but these interfaces are well-defined and should be easily adaptable. These command processors may be used with any operating system which supports the execution of configurations constructed with PCDs. In fact, such an operating system must use one of these (or a similar one) in order to build dynamic configurations.

The fourth objective was the exploration of the concept of user adaptable command processors. It is significant that the command processors for the NADEX operating system were designed to execute, not as an integral part of NADEX, but on top of the operating system. This concept allows flexibility in the creation and choice of command processors. Three very different command processors have been created. The NADEX user is free to choose any one of the three, tailor one to a particular need, or create a new one.

The Static command processor executes completely connected configurations. Although dynamically connected configurations may not be executed, any configuration that can be built at command interpretation time with UNIX or MIRACLE can also be constructed as a static configuration. This command processor does not need to connect ports or contain extra PCDs so it is relatively small and efficient.

The UNIX command processor is useful for creating dynamic linear configurations. It is best utilized for those "one of a kind" configurations that are not used enough to justify building a static configuration that could be used repeatedly. Its syntax is simple and much easier to use that of MIRACLE.

Dynamic configurations containing arbitrary graphs may be constructed with the MIRACLE command processor. With its control

structures and string functions, it is an extremely powerful interpreter, but its syntax is complicated because a great variety of symbols must be manipulated by the user.

Implementation

The Static, UNIX, and MIRACLE prototype command processors were developed in Sequential Pascal on the Interdata 8/32. Early testing was done under SOLEX, which was a static, relatively small version of NADEX used for development purposes. The dynamic connection of ports required the facilities of NADEX, under which all three command processors now run.

The Static and UNIX programs took approximately seven months to develop. A great deal of that time was spent in learning how to manipulate PCD records and interface with the file system and LINK program. During this time, the PCD record format changed twice and the entire system was under development so there was little or no documentation upon which to rely. Although the MIRACLE command processor implements a much more powerful language than either of its predecessors, its relatively short implementation time of two months reflects the experience gained previously.

With its strict type checking, powerful data structures, and self-documenting code, the choice of SPascal as the implementation language has had a favorable impact on the development of the command processors. The Symbolic Debugger was also very helpful, especially for tracking "phantom" errors that were a result of interface problems with LINK or the subsystems or when it was not clear where the error originated.

Command Processor	Lines of Code	Code Space	Data Space
Do	650	6K	2K
Static	2020	16K	6K
UNIX	2700	30K	26K
MIRACLE	3400	41K	32K

Figure 23. Comparison of command processors.

Brinch Hansen's command processor DO performs no parameter type checking: all parameters are passed to the appropriate programs without interpretation. This means that each program brought up by DO must parse its own parameters. The command processor is nice and small (see Figure 23), but much code for parsing parameters has to be duplicated by the called programs.

The NADEX command processors read in PCDs which contain templates for expected parameters and external ports. If a program uses a parameter that is a lexical pattern which does not match one of the command processor types, an array of 32 characters may be passed to the program for it to parse as it wishes. Otherwise, all arguments in the command line are parsed and compared with the templates. Programs are not called unless all ports are connected and their parameters are of the correct type, number, and order.

If any discrepancies are found such as parameters of the wrong type, missing or extra parameters, or incorrectly connected ports, an error message is displayed at the user's console. Mandatory and optional parameters are listed with their types and default values followed by external ports names (see the user sessions in Appendix C).

A deliberate attempt has been made to include descriptive error messages so that the user is given as much help as possible with ports and parameters. This is nice for the user, but not without its cost as

indicated by the increased code and data sizes of the NADEX command processors as listed in Figure 23. The command processor must reserve approximately 4600 bytes of data space for each PCD record. Since the Static command processor uses only one PCD at a time, its data space is small. Both UNIX and MIRACLE must reserve room for at least two PCDs--the PCD whose arguments are being parsed and the PCD being created for the command configuration.

The Static and UNIX command processors were bootstrapped off of DO and owe much of their structure and parsing routines to that program. They use a two-pass method of interpretation: the lexical and syntactic analysis is performed first, followed by a semantic analysis. Difficulties encountered with interpreting pipeline commands contributed to the use of a different approach in MIRACLE.

The MIRACLE command processor uses the same lexical routines, but its expression-oriented nature led to the use of modified operator-precedence parsing. Since it allows recursive evaluation of expressions, its data space is significantly larger than the other command processors. Further details of the implementions may be found in a technical report[5].

Further research

The command processors have yet to be vigorously tested by users other than those involved in the implementation. More user experience is needed to make the error messages as user-oriented as possible. MIRACLE presents the most complicated and as yet, untested user interface.

As mentioned in the Introduction, port mode and protocol information should perhaps be included in the help messages. Also, instead of abandoning a command when missing ports or parameters are detected, the command processor might prompt the user for the missing information. The original UNIX shell allows input and output bindings to default to the console, which also might be nice to implement with the NADEX command processors.

The UNIX shell handles many of the same control structures that are currently implemented in MIRACLE. If these prove to be valuable, they could be added to the NADEX version of UNIX. Other extensions to the languages, such as pattern matching, are also under investigation. Of particular importance are task control functions such as the MIRACLE's INIT, START, STOP, WAIT, KILL, etc. These have not yet been implemented as the status of an executing task is not currently available to a NADEX command processor.

UNIX and MIRACLE use a hierarchical file system[10] which is currently superimposed on the OS/32 MT "flat" file system. This can be rather confusing (a file created with OS/32 MT is not accessible to the hierarchical system until it has been manually added). The file system does not currently support the RENAME command.

Return codes from configurations which have been executed are not currently available to the command processor. Although the UNIX "andf" and "orf" and MIRACLE control structures depend on these return codes, they are currently implemented as always returning TRUE.

An interesting area of research lies with the graphical representation of configurations as input to a command processor. The user would theoretically use a light pen to "draw" the configuration to

be executed.

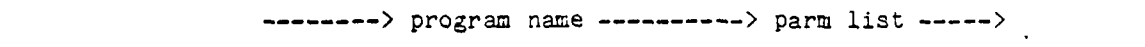
Since these command processors were developed as prototype models, small code size and efficiency were not emphasized during development. Perhaps their real success lies with the fact that both UNIX and MIRACLE are friendlier than the original systems.

APPENDIX A: SYNTAX GRAPHS

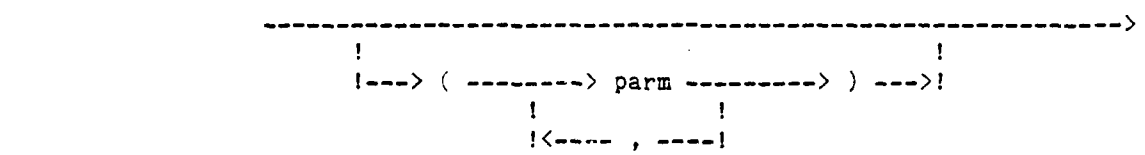
DO Syntax Graphs

1. command

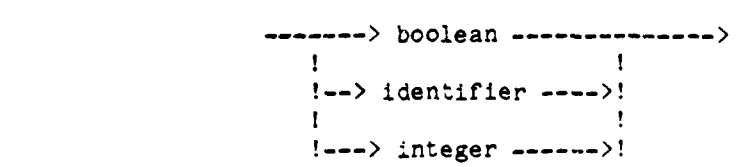
2. command statement



3. parm list



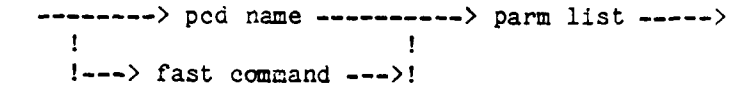
4. parm



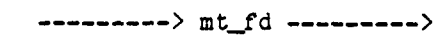
Static Syntax Graphs

1. command

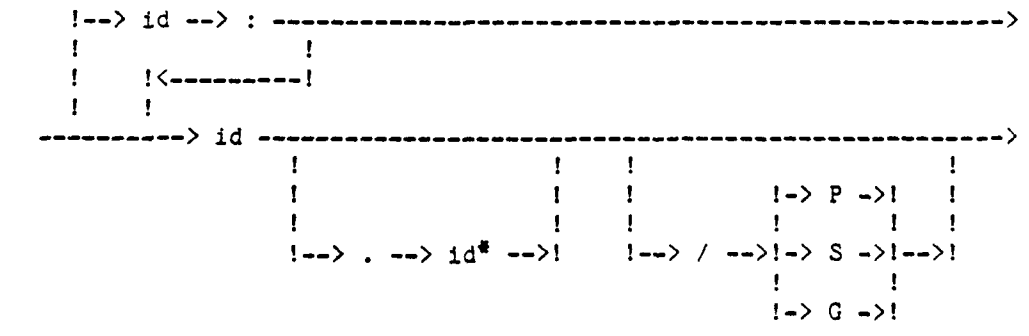
2. command statement



3. pcd_name

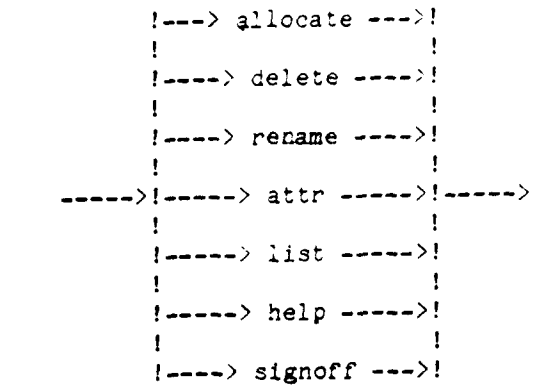


4. mt_fd

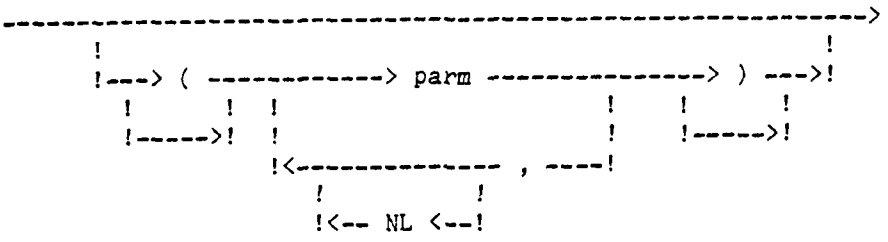


*The length of this identifier must be <= 3.

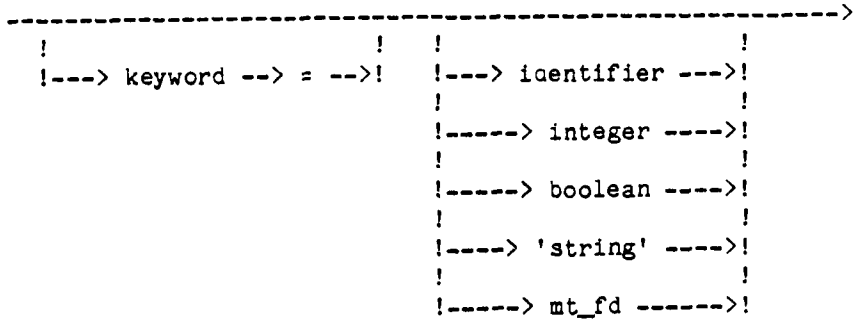
5. fast command



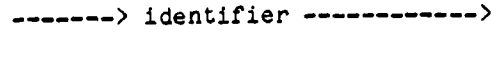
6. parm list



7. parm

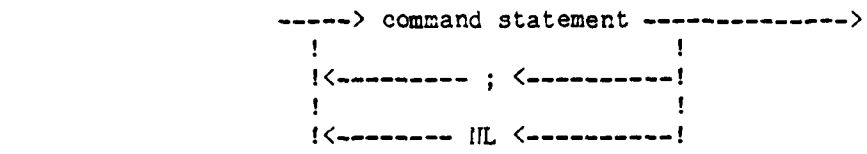


8. keyword

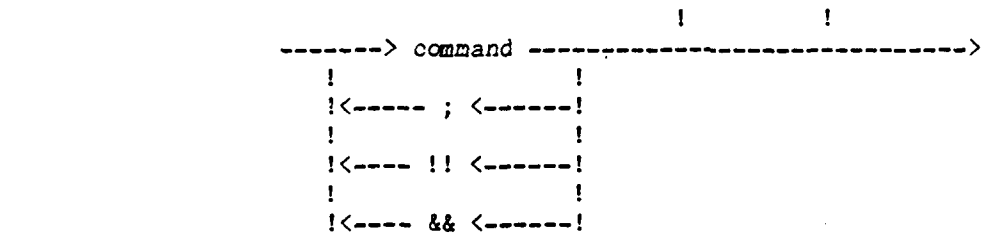


UNIX Syntax Graphs

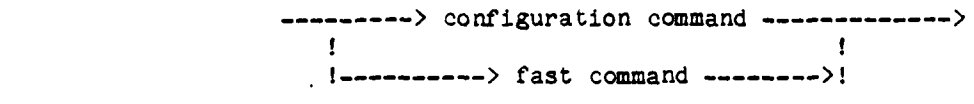
1. command statement list



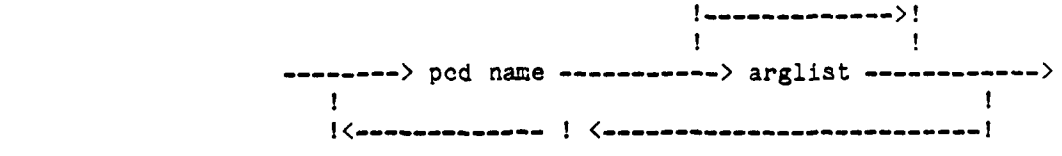
2. command statement



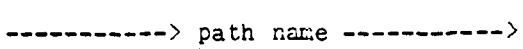
3. command



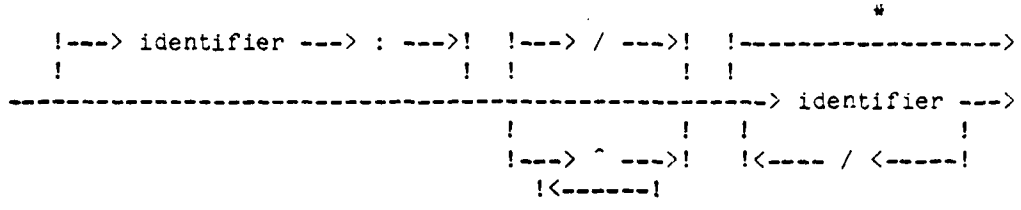
4. configuration command



5. pcd name

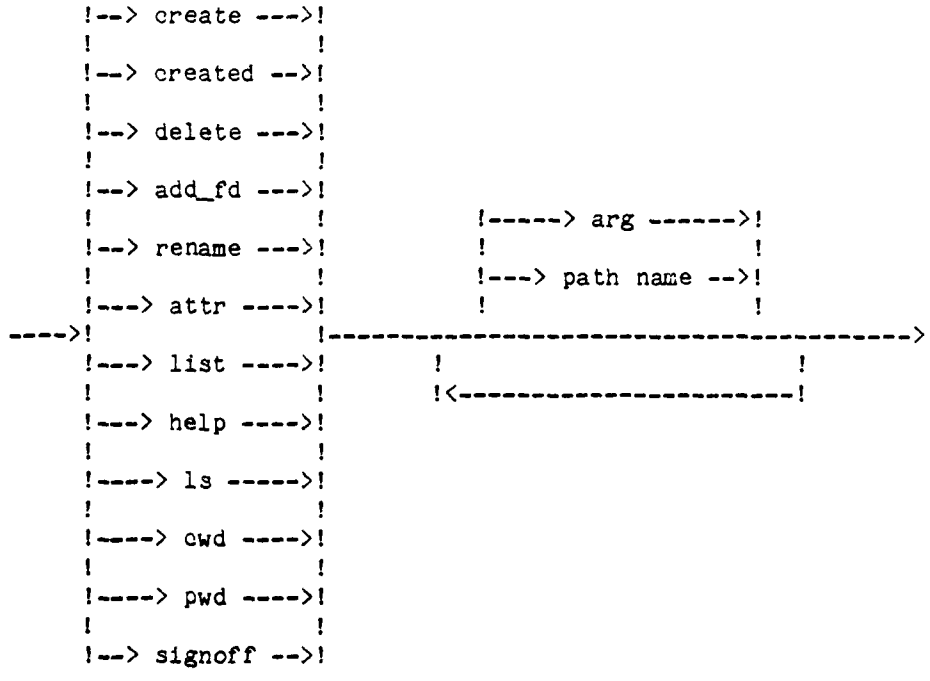


6. path name

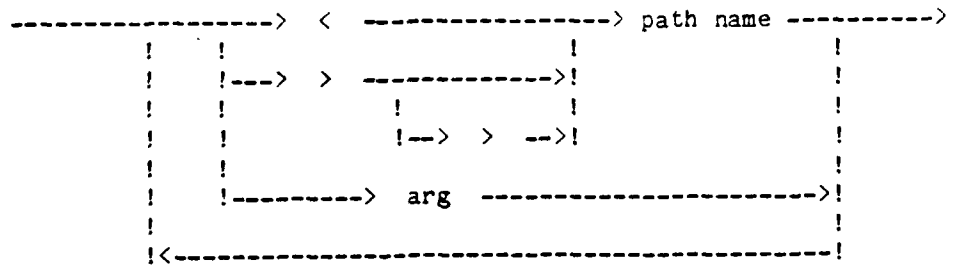


* this escape allowed only if path name <> null

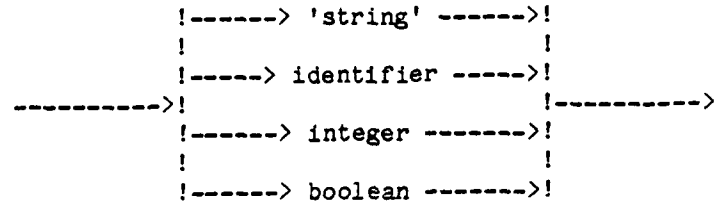
7. fast command



8. arglist

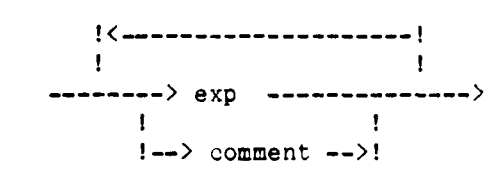


9. arg

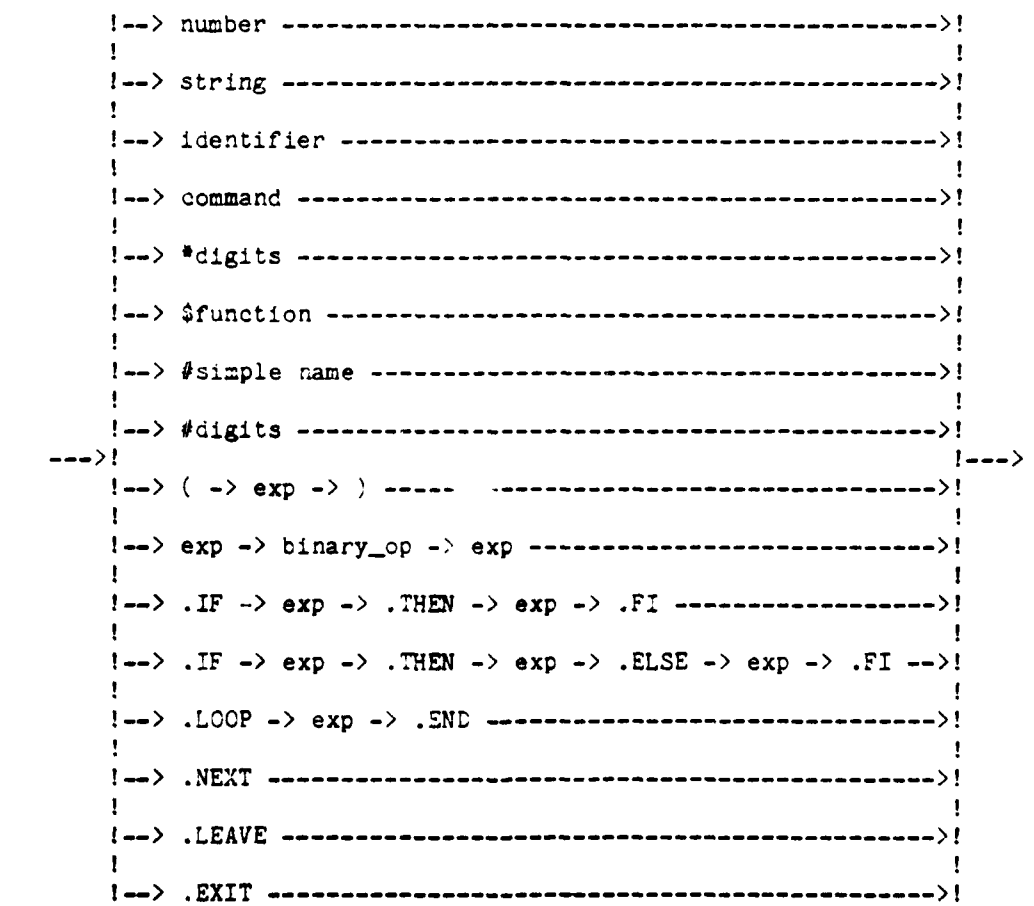


MIRACLE Syntax Graphs

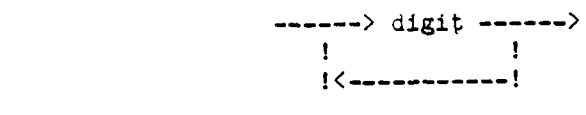
1. construct



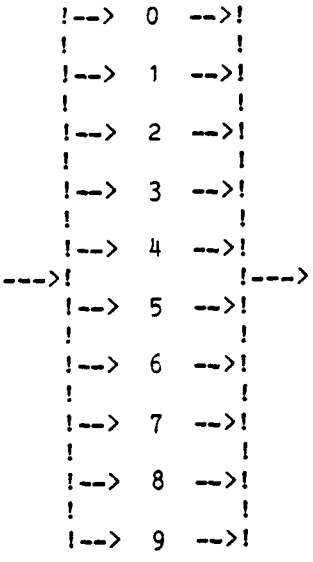
2. exp



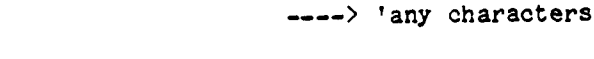
3. number



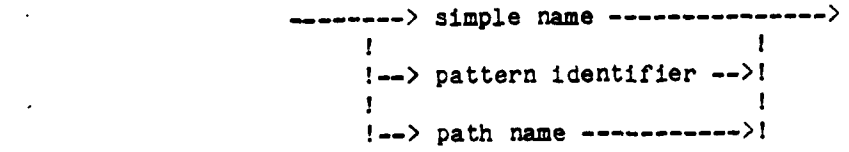
4. digit



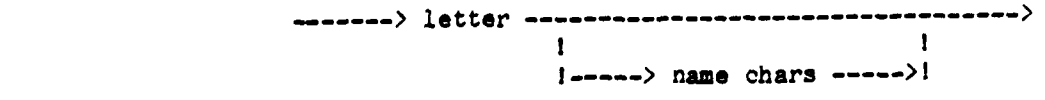
5. string



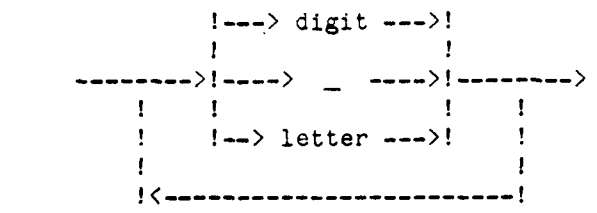
6. identifier



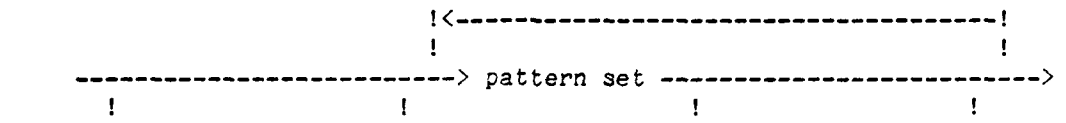
7. simple name



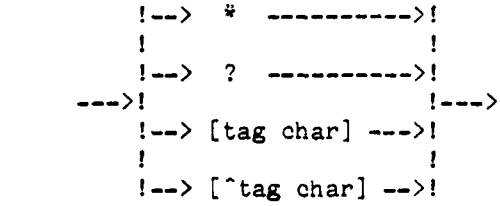
8. name chars



9. pattern identifier

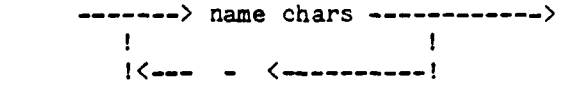


10. pattern set

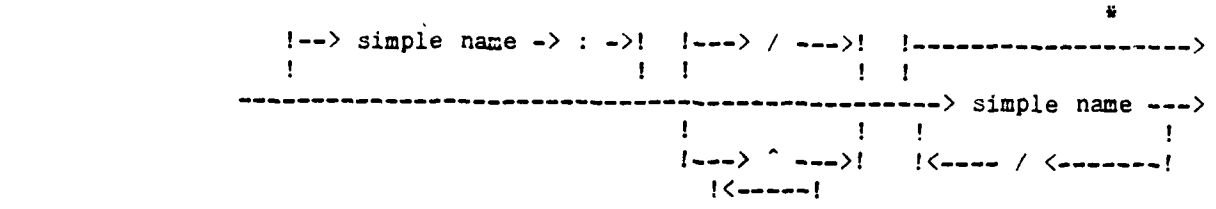


An asterisk matches any number of characters; the question mark matches a single character.

11. tag char

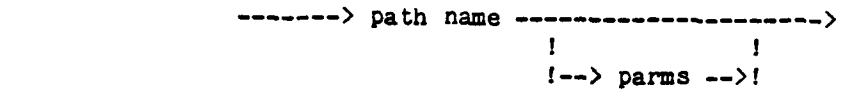


12. path name

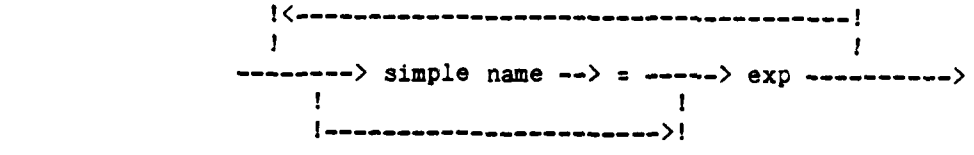


* this escape allowed only if path name <> null

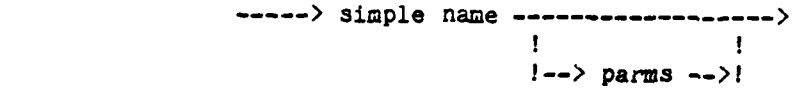
13. command



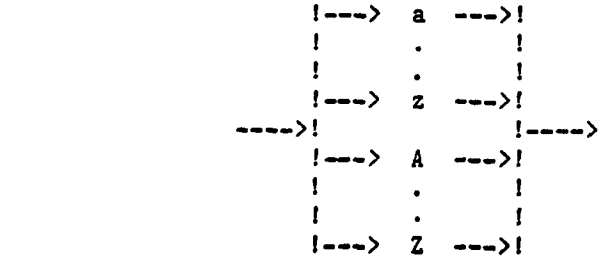
14. parms



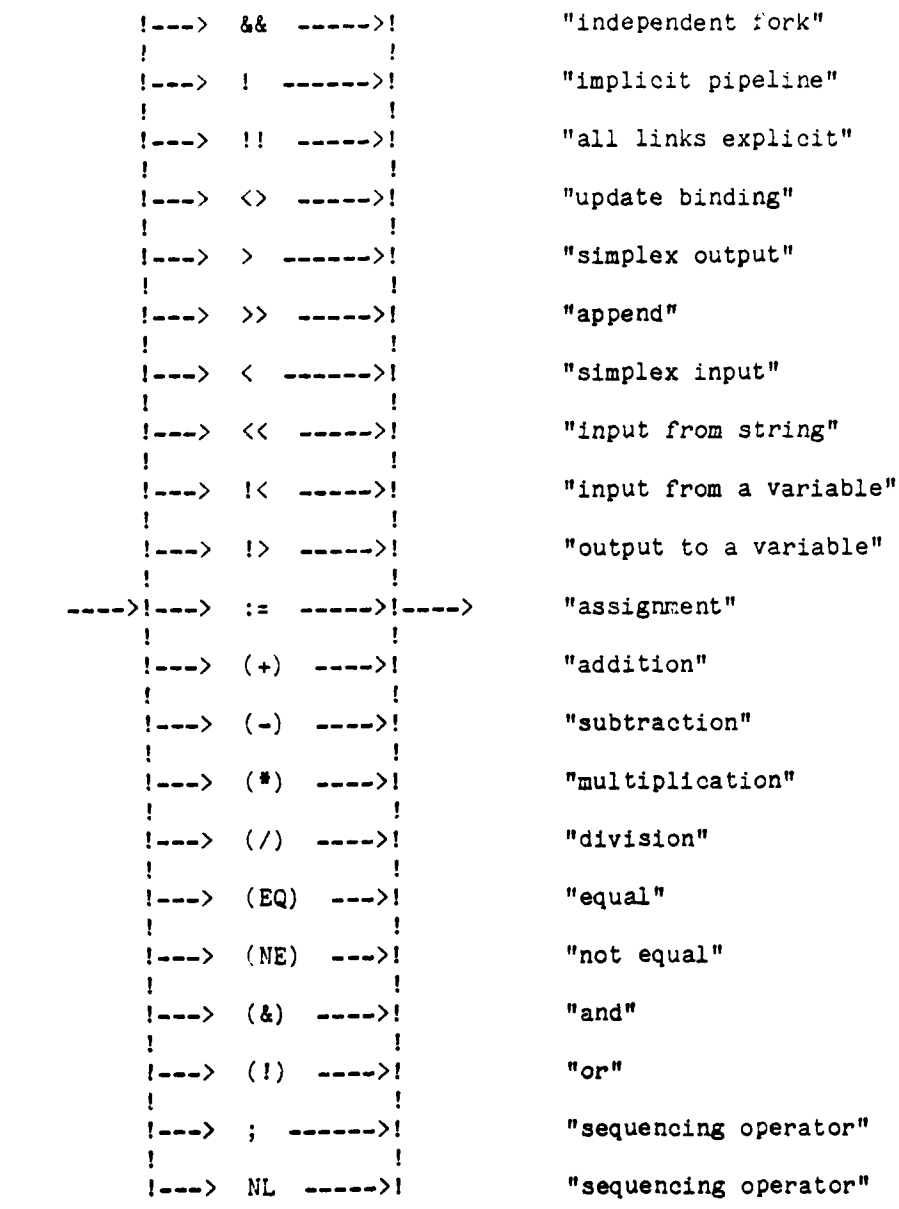
15. function



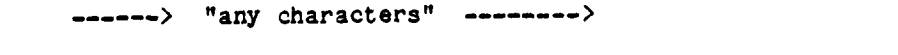
16. letter



17. binary_op



18. comment



APPENDIX B

```
"*****************************************************************
*                                                               *
*                        NADEX NATIVE PREFIX                    *
*                                                               *
*****************************************************************"

CONST PAGE_SIZE = 512;              "SIZE OF DATA PAGE"
      PARM_SIZE = 32;               "SIZE OF PARAMETER BLOCKS"
      MAX_DTS = 40;                 "MAX GLOBAL DTS ID"
      MAX_PORT = 20;                "MAX PORT ID"
      MAX_PARM = 10;                "MAX PARM ID"
      SVC1_BLOCK_SIZE = 24;         "SIZE OF SVC 1 PARM BLOCK"
      SVC7_BLOCK_SIZE = 28;         "SIZE OF SVC 7 PARM BLOCK"
      SD = 700;                     "PREFIX STACK DEPTH"

CONST NL = '(:10:)';      CR = '(:13:)';        ETB = '(:23:)';
      EM = '(:25:)';      BEL = '(:07:)';

TYPE PAGE = ARRAY [1..PAGE_SIZE] OF BYTE;
     PARAMETER = ARRAY [1..PARM_SIZE] OF BYTE;
     UNIV_SVC1_BLOCK = ARRAY [1..SVC1_BLOCK_SIZE] OF BYTE;
     UNIV_SVC7_BLOCK = ARRAY [1..SVC7_BLOCK_SIZE] OF BYTE;

TYPE DTS_INDX = 1..MAX_DTS;     DTS_INDX0 = 0..MAX_DTS;
     PORT_INDX = 1..MAX_PORT;   PORT_INDX0 = 0..MAX_PORT;
     PARM_INDX = 1..MAX_PARM;   PARM_INDX0 = 0..MAX_PARM;

TYPE DTS_SET = SET OF DTS_INDX;

TYPE BUF_TYPES = (PARM_BUF, DATA_BUF, NIL_BUF);   "BUFFER TYPES"

TYPE PREFIX_TYPES = (NATIVE_PREFIX "0", PASDRIVR_PREFIX "1");

TYPE REQ_CODES = (REQ_OK "0", REQ_NODE_ABORT "1", REQ_DTS_ABORT "2",
       REQ_DEFER "3", REQ_UNRES_DTS "4", REQ_PROT_ERROR "5",
       REQ_BAD_PORT "6");       "PREFIX DTS OPERATION RETURN CODES"

PROCEDURE READ_CHAR (PORT: PORT_INDX; VAR C:CHAR); SD;

PROCEDURE WRITE_CHAR (PORT: PORT_INDX; C:CHAR); SD;

PROCEDURE READ_DATA (PORT: PORT_INDX; VAR DATA: UNIV PAGE;
                     VAR LENGTH: INTEGER; VAR RESULT: REQ_CODES);
                     SD;
```

```
PROCEDURE WRITE_DATA (PORT: PORT_INDX; DATA: UNIV PAGE;
                      LENGTH: INTEGER; CONDITIONAL: BOOLEAN;
                      VAR RESULT: REQ_CODES); SD;

PROCEDURE READ_PARM (PORT: PORT_INDX; VAR PARM: UNIV PARAMETER;
                     VAR RESULT: REQ_CODES); SD;

PROCEDURE WRITE_PARM (PORT: PORT_INDX; PARM: UNIV PARAMETER;
                      CONDITIONAL: BOOLEAN; VAR RESULT: REQ_CODES);
                      SD;

PROCEDURE MAP_PORT (PORT: PORT_INDX; BUF_TYPE: BUF_TYPES;
                    VAR RDTS: DTS_INDX0; VAR WDTS: DTS_INDX0); SD;

PROCEDURE AWAIT_EVENTS (VAR READ_WAITS, WRITE_WAITS: DTS_SET;
                        VAR READ_READY, WRITE_READY: DTS_SET;
                        VAR RESULT: REQ_CODES); SD;

PROCEDURE DISCONNECT (PORT: PORT_INDX; VAR RESULT: REQ_CODES); SD;

PROCEDURE FETCH_USER_ATTRIBUTES; SD;

PROCEDURE SUBMIT_CONFIG; SD;

PROCEDURE SVC1 (VAR PARM: UNIV UNIV_SVC1_BLOCK); 64;
PROCEDURE SVC7 (VAR PARM: UNIV UNIV_SVC7_BLOCK); 64;

PROCEDURE FETCH_PARM (PARM_ID: PARM_INDX; VAR PARM: UNIV PARAMETER;
                      VAR OK: BOOLEAN); SD;

PROCEDURE LOAD_OVERLAY (PORT_ID: PORT_INDX; VAR OK: BOOLEAN); SD;

PROCEDURE INVOKE_OVERLAY (VAR ARG: INTEGER;
                          PREFIX_TYPE: PREFIX_TYPES;
                          VAR RESULT, LINENO: INTEGER); SD;

PROCEDURE CANCEL_NODE; SD;

PROCEDURE CANCEL_CONFIG; SD;

PROCEDURE BREAKPNT (LN: INTEGER); 64;
```

APPENDIX C

SAMPLE USER SESSIONS

```
NADEX PROTOTYPE FILE SUBSYSTEM                          ***************
NADEX PROTOTYPE LINKER R00-00                           *    STATIC   *
NADEX PROTOTYPE STATIC COMMAND PROCESSOR   R00-00       ***************

->HELP

THE FOLLOWING "FAST" COMMANDS ARE AVAILABLE:

    ALLOCATE  FD          "ALLOCATE A NEW FILE"
    DELETE    FD          "DELETE AN EXISTING FILE"
    ATTR      FD          "DISPLAY THE ATTRIBUTES OF A FILE"
    LIST      FD          "DISPLAY AN ASCII FILE AT THE CONSOLE"
    SIGNOFF               "TERMINATE SESSION"
    RENAME OLDFD,NEWFD    "RENAME A FILE"

THE FOLLOWING CONFIGURATIONS ARE AVAILABLE:

    PAS32             "RUNS SEQUENTIAL PASCAL COMPILER"
    UNIXLINK          "BRINGS UNIX CMDP AND LINK CONFIGURATION UP"
    MIRC              "BRINGS MIRACLE CMDP AND LINK UP"
    DUMP              "DISPLAYS A PCD"
    HI                "DEMONSTRATES A FRIENDLY SYSTEM"
    PCD               "BUILDS A NEW PCD"
    DINPHIL           "DINING PHILOSOPHERS CONFIGURATION"
    SIMTEST           "SIMULATION CONFIGURATION"

    ENTERING THE NAME OF ANY COMMAND OR CONFIGURATION
         WILL DISPLAY ITS CALLING FORMAT.

->ALLOCATE NEW.TXT
->ATTR NEW.TXT

FILE  SYS2:NEW.TXT/P  HAS   0 RECORDS OF LENGTH  512
->RENAME NEW.TXT,OLD.TXT
->DELETE OLD.TXT
->ATTR OLD.TXT

FILE NAME ERROR:  SYS2:OLD.TXT/P
```

```
->TEST

MANDATORY PARAMETER OMITTED

   TRY AGAIN:

        TEST        INFILE  : FD,
                   [PROGFILE: FD DEFAULT(SYS2:GREP.IMG/P)],
                   [CODESP  : INT DEFAULT(5)],
                   [DATASP  : INT DEFAULT(10)],
                   [OLAYSP  : INT DEFAULT(0)],
                   [INT     : INT DEFAULT(1111)],
                   [BOOL    : BOOL DEFAULT(TRUE)],
                   [STRING  : STRING DEFAULT('HI THERE')],
                   [ID      : ID DEFAULT(ID)],
                   [FILENAME: FD DEFAULT(SYS2:FILENAME.TXT/P)]

->**ILLEGAL$$CHARACTERS???
TRY AGAIN:
   CONFIGURATION_NAME
OR
   CONFIGURATION_NAME         [ KEYWORD = ] ARG, ... ,
                         ..., [ KEYWORD = ] ARG
USING
 ARG: BOOLEAN, INTEGER, IDENTIFIER, STRING, FILE_DESCRIPTOR, OR ARGLIST
->UNIXLINK
NADEX PROTOTYPE LINKER R00-00                   *************
WORKING DIRECTORY: /HELP                        *   UNIX    *
UNIX PROTOTYPE COMMAND PROCESSOR   R00-00       *************
->HELP
THE FOLLOWING "FAST" COMMANDS ARE AVAILABLE:
    LS                   "LIST FILE NAMES IN THE CURRENT DIRECTORY"
    PWD                  "PRINT CURRENT WORKING DIRECTORY"
    CWD        PN        "CHANGE WORKING DIRECTORY TO PATH NAME"
    CREATE     PN        "CREATE A NEW ASCII FILE"
    CREATED    PN        "CREATE A NEW DIRECTORY FILE"
    DELETE     PN        "DELETE AN EXISTING FILE"
    ATTR       PN        "DISPLAY THE ATTRIBUTES OF A FILE"
    LIST       PN        "DISPLAY AN ASCII FILE AT THE CONSOLE"
    ADD_FD PN 'MT_FD'    "ADD THE MT_FD TO THE CURRENT DIRECTORY"
    SIGNOFF              "TERMINATE SESSION"

THE FOLLOWING CONFIGURATIONS ARE AVAILABLE:
       PAS32          "RUNS SEQUENTIAL PASCAL COMPILER"
       MIRC           "BRINGS MIRACLE CMDP AND LINK CONFIGURATION UP"
       COPY           "COPIES ASCII FILES"
       WC             "PRINTS NO. OF CHARACTERS, WORDS, AND LINES"
       GREP           "SELECTS LINES CONTAINING SPECIFIED STRING"
       HI             "DEMONSTRATES A FRIENDLY SYSTEM"
       PCD            "ALLOWS USER TO CREATE NEW PCDS"
       UDUMP          "DISPLAYS A PCD AT THE CONSOLE"
    ENTERING THE NAME OF ANY COMMAND OR CONFIGURATION
         WILL DISPLAY ITS CALLING FORMAT.
```

```
->$HELP

TRY AGAIN
     PCD_NAME
OR
 PCD_NAME [ARG ARG...] [<PN ] [! PCD_NAME [ARG ARG ...] ]  [ >PN ]  [&]
USING
    ARG: BOOLEAN, INTEGER, IDENTIFIER, STRING, PATH NAME, OR ARGLIST
->CWD /KIM
WORKING DIRECTORY: /KIM
->LS
ENTRIES IN DIRECTORY  /KIM
     ^             SYS2:U0000000.DIR/P
     CONWAY        SYS2:CONWAY.TXT/P
     PCDS          SYS2:U000001D.DIR/P
     LINK          SYS2:U000002D.DIR/P
     UFSS          SYS2:U000002E.DIR/P
     PCD           SYS2:U000002F.DIR/P
     DUMP          SYS2:U0000030.DIR/P
     CSS           SYS2:U0000033.DIR/P
     IMG           SYS2:U0000038.DIR/P
     PBM           SYS2:U0000036.DIR/P
     LS            SYS2:U0000044.DIR/P
     ED            SYS2:U0000049.DAT/P

->CWD ^ ;
WORKING DIRECTORY: /
->CWD /HELP; LS
WORKING DIRECTORY: /HELP
ENTRIES IN DIRECTORY  /HELP
     ^             SYS2:U0000000.DIR/P
     FILE          SYS2:U0000039.DAT/P
     UNIX          SYS2:UNIXHELP.TXT/P
     CMDP          SYS2:CMDPHELP.TXT/P

->ALLOCATE NEWFILE
FILE NAME ERROR:  SYS2:ALLOCATE.PCD/P
->CREATE NEWFILE
SYS2:U0000050./P
->ATTR NEWFILE

FILE  SYS2:U0000050.DAT/P  HAS   0 RECORDS OF LENGTH  512
->LS
ENTRIES IN DIRECTORY  /HELP
     ^             SYS2:U0000000.DIR/P
     FILE          SYS2:U0000039.DAT/P
     UNIX          SYS2:UNIXHELP.TXT/P
     CMDP          SYS2:CMDPHELP.TXT/P
     NEWFILE       SYS2:U0000050.DAT/P
```

```
->COPY

ERROR IN PORT CONNECTIONS:
     COPY1.IN, COPY1.OUT

->COPY <UNIXHELP >NEWFILE

PATHNAME ERROR:  UNIXHELP

   TRY AGAIN:

        COPY
           EXTERNAL PORTS: IN, OUT

->COPY <UNIX >NEWFILE
NADEX PROTOTYPE LINKER R00-00
WORKING DIRECTORY: /HELP
UNIX PROTOTYPE COMMAND PROCESSOR   R00-00
->LIAST NEWFILE
FILE NAME ERROR:  SYS2:LIAST.PCD/P
->LIST NEWFILE
THE FOLLOWING "FAST" COMMANDS ARE AVAILABLE:
    LS                  "LIST FILE NAMES IN THE CURRENT DIRECTORY"
    PWD                 "PRINT CURRENT WORKING DIRECTORY"
    CWD        PN       "CHANGE WORKING DIRECTORY TO PATH NAME"
    CREATE     PN       "CREATE A NEW ASCII FILE"
    CREATED    PN       "CREATE A NEW DIRECTORY FILE"
    DELETE     PN       "DELETE AN EXISTING FILE"
    ATTR       PN       "DISPLAY THE ATTRIBUTES OF A FILE"
    LIST       PN       "DISPLAY AN ASCII FILE AT THE CONSOLE"
    ADD_FD PN 'MT_FD'   "ADD THE MT_FD TO THE CURRENT DIRECTORY"
    SIGNOFF             "TERMINATE SESSION"

THE FOLLOWING CONFIGURATIONS ARE AVAILABLE:
       PAS32          "RUNS SEQUENTIAL PASCAL COMPILER"
       MIRC           "BRINGS MIRACLE CMDP AND LINK CONFIGURATION UP"
       COPY           "COPIES ASCII FILES"
       WC             "PRINTS NO. OF CHARACTERS, WORDS, AND LINES"
       GREP           "SELECTS LINES CONTAINING SPECIFIED STRING"
       HI             "DEMONSTRATES A FRIENDLY SYSTEM"
       PCD            "ALLOWS USER TO CREATE NEW PCDS"
       UDUMP          "DISPLAYS A PCD AT THE CONSOLE"
    ENTERING THE NAME OF ANY COMMAND OR CONFIGURATION
        WILL DISPLAY ITS CALLING FORMAT.

->ATTR NEWFILE

FILE  SYS2:U0000050.DAT/P  HAS   3 RECORDS OF LENGTH  512
```

```
->GREP

MANDATORY PARAMETER OMITTED

   TRY AGAIN:

        GREP        SOUGHT  : STRING
           EXTERNAL PORTS: IN, OUT

->GREP 'ALL' <NEWFILE ! CONSOLE  &&  LS
      PCD         "ALLOWS USER TO CREATE NEW PCDS"
         WILL DISPLAY ITS CALLING FORMAT.

NADEX PROTOTYPE LINKER R00-00
WORKING DIRECTORY: /HELP
UNIX PROTOTYPE COMMAND PROCESSOR   R00-00
ENTRIES IN DIRECTORY  /HELP
     .            SYS2:U0000000.DIR/P
     FILE         SYS2:U0000039.DAT/P
     UNIX         SYS2:UNIXHELP.TXT/P
     CMDP         SYS2:CMDPHELP.TXT/P

->GREP 'AL' <NEWFILE ! WC ! CONSOLE
58
12
 2

NADEX PROTOTYPE LINKER R00-00
WORKING DIRECTORY: /HELP
UNIX PROTOTYPE COMMAND PROCESSOR   R00-00
->GREP 'AL' <NEWFILE ! GREP 'US' ! WC ! WC ! CONSOLE  &  HI
 4
 3
 2

* * * * * * * *  HELLO YOURSELF  * * * * * * * *
NADEX PROTOTYPE LINKER R00-00
WORKING DIRECTORY: /HELP
UNIX PROTOTYPE COMMAND PROCESSOR   R00-00
->GREP 'AL <NEWFILE ! WC

STRING TOO LONG.

TRY AGAIN
     PCD_NAME
OR
 PCD_NAME [ARG ARG...] [<FD ] [! PCD_NAME [ARG ARG ...] ]  [ >FD ]  [&]
USING
 ARG: BOOLEAN, INTEGER, IDENTIFIER, STRING, FILE_DESCRIPTOR, OR ARGLIST
->GREP 'AL' <NEWFILE ! WC

ERROR IN PORT CONNECTIONS:
     WC2.OUT
```

```
->GREP ! WC

MANDATORY PARAMETER OMITTED

   TRY AGAIN:

        GREP        SOUGHT  : STRING
           EXTERNAL PORTS: IN, OUT

->MIRC
NADEX PROTOTYPE LINKER R00-00                    ***************
WORKING DIRECTORY: /KIM/IMG                      *   MIRACLE   *
MIRACLE PROTOTYPE COMMAND PROCESSOR   R00-00     ***************

->$HELP
THE FOLLOWING "FAST" COMMANDS ARE AVAILABLE:

   $LS                  "LIST FILE NAMES IN THE CURRENT DIRECTORY"
   $PWD                 "PRINT CURRENT WORKING DIRECTORY"
   $CWD        PN       "CHANGE WORKING DIRECTORY TO PATH NAME"
   $ALLOC      PN       "ALLOCATE A NEW ASCII FILE"
   $ALLOCD     PN       "ALLOCATE A NEW DIRECTORY FILE"
   $DELETE     PN       "DELETE AN EXISTING FILE"
   $ATTR       PN       "DISPLAY THE ATTRIBUTES OF A FILE"
   $LIST       PN       "DISPLAY AN ASCII FILE AT THE CONSOLE"
   $EDIT $INT_CMDFILE   "DISPLAY AN INTERNAL COMMAND FILE"
   $DCL   VAR: TYPE     "DECLARE A VARIABLE NAME AND ITS TYPE"
   $NOT       EXP       "NEGATES THE BOOLEAN RESULT OF THE EXP"
   $ADD_FD PN 'MT_FD'   "ADD THE MT_FD TO THE CURRENT DIRECTORY"
   $SIGNOFF             "TERMINATE SESSION"

->HELP
THE FOLLOWING CONFIGURATIONS ARE AVAILABLE:

       PAS32          "RUNS SEQUENTIAL PASCAL COMPILER"
       MIRC           "BRINGS MIRACLE CMDP AND LINK CONFIGURATION UP"
       COPY           "COPIES ASCII FILES"
       WC             "PRINTS NO. OF CHARACTERS, WORDS, AND LINES"
       GREP           "SELECTS LINES CONTAINING SPECIFIED STRING"
       HI             "DEMONSTRATES A FRIENDLY SYSTEM"
       PCD            "ALLOWS USER TO CREATE NEW PCDS"
       UDUMP          "DISPLAYS A PCD AT THE CONSOLE"
       DINPHIL        "DINING PHILOSOPHERS CONFIGURATION"
       SIMTEST        "SIMULATION CONFIGURATION"

    ENTERING THE NAME OF ANY COMMAND OR CONFIGURATION
        WILL DISPLAY ITS CALLING FORMAT.

TYPE $HELP FOR FAST COMMANDS HELP
```

```
->$ALLOCD /KIM/NEWDR
SYS2:U0000055./P
ZERO PAGES IN DIRECTORY: /KIM/NEWDR
->$CWD /KIM/NEWDR
WORKING DIRECTORY: /KIM/NEWDR
->ADD_FD
TRY AGAIN:   $ADD_FD PATHNAME 'MT_FD'
->$ADD_FD NEWFILE 'NEWFILE.PAS'
->$LS
ENTRIES IN DIRECTORY  /KIM/NEWDR
      ^             SYS2:U000001C.DIR/P
     NEWFILE        SYS2:NEWFILE.PAS/P

->$ATTR NEWFILE

FILE  SYS2:NEWFILE.PAS/P  HAS   0 RECORDS OF LENGTH  512
->HI && HI
* * * * * * * *  HELLO YOURSELF  * * * * * * * *
* * * * * * * *  HELLO YOURSELF  * * * * * * * *
NADEX PROTOTYPE LINKER R00-00
WORKING DIRECTORY: /KIM/NEWDR
MIRACLE PROTOTYPE COMMAND PROCESSOR   R00-00
->.IF TRUE .ELSE $LS .FI
MISSING .THEN
->.IF FALSE
->$LS .THEN $PWD .ELSE $CWD /HELP .FI
ENTRIES IN DIRECTORY  /KIM/NEWDR
      ^             SYS2:U000001C.DIR/P
     NEWFILE        SYS2:NEWFILE.PAS/P
/KIM/NEWDR

->CWD ^IMG
WORKING DIRECTORY: /KIM/IMG
->$PWD

/KIM/IMG
->$PWD;
/KIM/IMG

SEQUENCING OPERAND ERROR
->$ALLOC 1

TRY AGAIN:   $ALLOC PATHNAME
->$LS
ENTRIES IN DIRECTORY  /KIM/IMG
      ^             SYS2:U000001C.DIR/P
     GREP           SYS2:GREP.IMG/P
     WC             SYS2:WC.IMG/P
     DUMP           SYS2:DCDF.IMG/P
     PEDIT          SYS2:PEDIT.IMG/P
     MASTER         SYS2:MASTER.IMG/P
     JOB1TEST       SYS2:JOB1TEST.IMG/P
     JOB2TEST       SYS2:JOB2TEST.IMG/P
```

```
->$ALLOC /KIM/IMG/ALLOCFILE
SYS2:U0000056./P
->$ATTR ALLOCFILE

FILE  SYS2:U0000056.DAT/P  HAS   0 RECORDS OF LENGTH  512
->$DELETE ALLOCFILE
->$ATTR ALLOCFILE
FILE NOT FOUND

PATHNAME ERROR:  ALLOCFILE
->WC

ERROR IN PORT CONNECTIONS:
     WC.IN, WC.OUT

->WC <FILE >OUTFILE

MISSING PORT NAME

ERROR IN PORT CONNECTIONS:
     WC.IN, WC.OUT

->.IF TRUE .THEN $PWD .FI

/KIM/IMG
->$EDIT $SIMSCEN
SIMMON MASTER<>*1 SCENE1<>*2 SCENE2<>*3 !!  SCENARIO MASTER SIMMON<>*1 !
! SCENARIO JOB1TEST SIMMON<>*2 !!  SCENARIO JOB2TEST SIMMON<>*3

->$EDIT $IFPARM

FILE NAME ERROR:  SYS2:IFPARM.STR/P
->$EDIT $IFPWD
 .IF $PWD .THEN $PWD .FI
->$IFPWD
/KIM/IMG
/KIM/IMG
->$FN4 := '$LS'
->$FN4
ENTRIES IN DIRECTORY  /KIM/IMG
     ^            SYS2:U0000010.DIR/P
     GREP         SYS2:GREP.IMG/P
     WC           SYS2:WC.IMG/P
     DUMP         SYS2:DCDF.IMG/P
     PEDIT        SYS2:PEDIT.IMG/P
     MASTER       SYS2:MASTER.IMG/P
     JOB1TEST     SYS2:JOB1TEST.IMG/P
     JOB2TEST     SYS2:JOB2TEST.IMG/P
```

```
->.IF $NOT ($ATTR PAS32) .THEN $PWD .ELSE $LS .FI
FILE NOT FOUND

PATHNAME ERROR:  PAS32
/KIM/IMG

->.IF TRUE .THEN .IF $PWD .THEN $LS .ELSE $CWD /KIM/PCD .FI .FI
/KIM/IMG
ENTRIES IN DIRECTORY  /KIM/IMG
     ^             SYS2:U000001C.DIR/P
     GREP          SYS2:GREP.IMG/P
     WC            SYS2:WC.IMG/P
     DUMP          SYS2:DCDF.IMG/P
     PEDIT         SYS2:PEDIT.IMG/P
     MASTER        SYS2:MASTER.IMG/P
     JOB1TEST      SYS2:JOB1TEST.IMG/P
     JOB2TEST      SYS2:JOB2TEST.IMG/P

->$SIGNOFF
```

REFERENCES

1. Balzer, R. M. Ports--A method for dynamic interprogram communication and job control. Proc. AFIPS Spring Joint Computer Conference 38 (1971), 485-489.

2. Bourne, S. R. The UNIX shell. The Bell System Technical Journal 57, 6, Part 2 (July-August 1978), 1971-1990.

3. Brinch Hansen, Per. The Architecture of Concurrent Programs. Prentice Hall, Englewood Cliffs, New Jersey, 1977.

4. Fundis, R. M., and Wallentine, Virgil. NADEX command processors implementation, Technical Report TR-80-03, Department of Computer Science, Kansas State University, Manhattan, Ks., July 1980.

5. Gray, T. E. Network job control: the tower of Babel revisited. Doctoral Dissertation, UCLA, March 1979.

6. Hoare, C. A. R. Communicating sequential processes. Communications of the ACM 21, 8 (August 1978), 666-677.

7. Jones, Anita K., et. al. StarOS, a multiprocessor operating system for the support of task forces. Proceedings of the Seventh Symposium on Operating System Principles, ACM SIGOPS (December 1979), 117-127.

8. Ousterhout, John K., et. al. Medusa: an experiment in distributed operating system structure, Communications of the ACM 23, 2, (February 1980), 92-104.

9. Rochat, Kim. A software structuring tool for message-based systems. M.S. Thesis, Dept. of Computer Science, Kansas State University, Manhattan, Ks., 1980.

10. Rochat, Kim, and Wallentine, Virgil. NADEX job control system implementation, Technical Report TR-80-05, Department of Computer Science, Kansas State University, Manhattan, Ks., May 1980.

11. Young, Robert, and Wallentine, Virgil. The NADEX core operating system services, Technical Report TR-79-11, Department of Computer Science, Kansas State University, Manhattan, Ks., November 1979.

END

DATE
FILMED
10-81

DTIC